

ARM 嵌入式系统开发

——软件设计与优化

ARM System Developer's Guide:
Designing and Optimizing System Software

[美] Andrew N. Sloss

[英] Dominic Symes 著

[美] Chris Wright

沈建华 译

北京航空航天大学出版社

内容简介

本书从软件设计的角度,全面、系统地介绍了 ARM 处理器的基本体系结构和软件设计与优化方法。内容包括:ARM 处理器基础;ARM/Thumb 指令集;C 语言与汇编语言程序的设计与优化;基本运算、操作的优化;基于 ARM 的 DSP;异常与中断处理;固件与嵌入式 OS;cache 与存储器管理;ARMv6 体系结构的特点等。全书内容完整,针对各种不同的 ARM 内核系统结构都有详尽论述,并有大量的例子和源代码。附录给出了完整的 ARMv4/v5/Thumb 指令的功能、编码、周期定时以及汇编参考。

本书适于从事 ARM 嵌入式系统教学与研发,或想把其它嵌入式平台的软件移植到 ARM 平台上去的专业技术人员使用,要求对 ARM 处理器有一定的了解,并有 C 语言和汇编语言基础。若在编译原理、操作系统、数字信号处理、计算机体系结构等方面有一定的基础,则效果会更好。本书也可作为嵌入式系统专业方向的本科生和研究生相关课程的教材或教学参考书。

图书在版编目(CIP)数据

ARM 嵌入式系统开发:软件设计与优化/(美)斯洛斯(Sloss, A. N.)等著;沈建华译. — 北京:北京航空航天大学出版社,2005.5

书名原文:ARM System Developer's Guide: Designing and Optimizing System Software

ISBN 7-81077-652-5

I. A… II. ①斯…②沈… III. 微处理器,ARM—系统设计 IV. TP332

中国版本图书馆 CIP 数据核字(2005)第 023570 号

ARM 嵌入式系统开发——软件设计与优化

ARM System Developer's Guide:

Designing and Optimizing System Software

[美] Andrew N. Sloss

[英] Dominic Symes 著

[美] Chris Wright

沈建华 译

责任编辑 王 瑛

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100083) 发行部电话:(010)82317024 传真:(010)82328026

<http://www.buaapress.com.cn> E-mail: bhpress@263.net

涿州市新华印刷有限公司印装 各地书店经销

*

开本:787 mm×960 mm 1/16 印张:41.75 字数:935 千字

2005 年 5 月第 1 版 2005 年 5 月第 1 次印刷 印数:5 000 册

ISBN 7-81077-652-5 定价:75.00 元

版 权 声 明

北京市版权局著作权登记号： 图字：01 - 2004 - 4607

ARM System Developers Guide: Designing and Optimizing System Software

Andrew Sloss

ISBN: 1-55860-874-5

Copyright © 2004 by Elsevier. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 981-2591-31-8

Copyright © 2005 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Elsevier (Singapore) Pte Ltd.

3 Killiney Road

#08-01 Winsland House I

Singapore 239519

Tel: (65) 6349-0200

Fax: (65) 6733-1817

First Published 2005

2005 年初版

Printed in China by Beijing University of Aeronautics and Astronautics Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由北京航空航天大学出版社与 Elsevier (Singapore) Pte Ltd. 在中国大陆境内合作出版。本版仅限在中国境内(不包括香港特别行政区及台湾地区)出版及标价销售。未经许可之出口,视为违反著作权法,将受法律之制裁。

致中国读者

首先我们向所有中国读者问好。

很高兴能出版我们书的中文版,希望你们有许多时间来学习 ARM 技术。如果你是位教授,那么这本书会有助于把 ARM 处理器内核和重要的软件编程技术介绍给学生;如果你是位学生,那么这本书是学习 ARM 编程的好读物;如果你是位使用 ARM 核进行产品研发的工程师,那么这本书可以满足实际需要。

此书的目的是展示如何设计与优化基于 ARM 处理器核的系统软件。我们的目标是提供一个基础的工作参考,以便能利用它开发出许多成功的新产品。

2004 年,ARM 的合作伙伴生产制造了 12 亿片 ARM 处理器,ARM 也推出了 2 个新的处理器:Cortex - M3 和 MPCore。Cortex - M3 主要针对微控制器市场,而 MPCore 主要针对高端的消费类产品。

Cortex - M3 改进了代码密度,减少了中断延迟,并有更低的功耗。Cortex - M3 实现了本书第 15 章所提到的 Thumb - 2 指令集。MPCore 是第一个 ARM 多处理器内核,执行基于均衡多处理器(SMP)的操作系统。MPCore 提供了 cache 一致性,每个实现支持 1~4 个 ARM11 核,这种设计为现代消费类产品对性能和功耗的需求作了很好的平衡。ARM 还引入了 L2 cache 控制器来改进系统的整体性能。

为了支持大量的密集数据处理,ARM 引入了 OptimoDE 技术,把它作为一个可配置的、专用的超长指令字(VLIW)数据引擎,配合 ARM 处理器工作。这种数据引擎可帮助主处理器完成诸如 MPEG4 或 H. 264 等复杂算法。

ARM 嵌入式系统开发

ARM 的大学计划将继续促进在应用研究、大学生和研究生课程以及教学培训中使用 ARM 技术。ARM 已经给那些从事片上系统(SoC)设计教学和研究的大学,授权了开发工具、仿真模型和物理 IP,并与剑桥(Cambridge)、卡内基·梅隆(Carnegie-Mellon)、密西根(Michigan)等大学保持密切联系。

ARM 的研究和开发机构正专注于改进代码密度、降低功耗、扩展多媒体功能以及提高安全性。

最后,我们要对沈教授表示最深切的感谢,为他在此书中文版翻译工作中的专注与努力。

谢谢您,沈教授。

Andrew N. Sloss

Dominic Symes

Chris Wright

2005 年 2 月

译者序

2004年7月,一个偶然的机,我去北京参加美国国家半导体(NS)公司的一个颁奖活动,正巧北京航空航天大学出版社的马广云老师打电话给我,说有一本关于ARM软件设计与优化的英文原版书,各方评价都很高,是否可以帮助翻译。我看到这本原版著作后,便当即答应尽快翻译此书。这确实是一本好书。

我是从2000年开始接触、使用ARM处理器的。2001年2月至2002年2月,我在加拿大维多利亚大学访学期间,对ARM体系结构进行了分析、研究,并以Cirrus Logic的EP7312和EP9312为美国一家公司做了两个嵌入式无线应用的系统设计。出于对嵌入式系统的浓厚兴趣和对ARM处理器前景的认同,近几年,我和我们实验室的其他老师、研究生一起,使用ARM处理器设计、开发了10多个应用项目、产品和一些软/硬件基础平台。软件部分包括OS移植、驱动、FAT文件系统、GUI、TCP/IP网络协议栈、WiFi(802.11b)、USB协议栈、实时音频、图像信号处理以及许多应用软件。涉及的ARM处理器芯片有Samsung的S3C44B0X,4510,2410;Atmel的AT91R40008, RM9200; Cirrus Logic的EP7312, EP9312; Philips的LPC2106, 2132, 2292; OKI的ML674000; Analog Device的AduC7026; Intel的Xscale PXA255等。

多年对ARM系统的研究与实践,一方面加深了我们对ARM处理器系统的理解并丰富了我们的经验;另一方面也使我们深深感到了嵌入式软件设计与优化的特殊重要性。嵌入式系统由于其硬件资源、成本的限制,以及一些实时需求,许多在PC等标准平台上的软件(包括系统软件和应用软件)都不能直接搬到一个特殊的嵌入式硬件平台上,而必须经过适当的裁剪、优化;否则就可能产生很差的效果,甚至会导致系统瘫痪。通过对软件

ARM 嵌入式系统开发

系统的合理设计与优化,可以降低对系统硬件资源(如 CPU 性能、存储器容量等)的需求,也可以降低系统功耗,从而使一个不可行的系统变为可行,也可以把一个毫无竞争力的产品变得极有竞争力!这里,可以把程序的优化过程看作是一个艺术加工过程,精雕细琢,一步步把一个平庸之作变成一件艺术品,体现出其真正的价值。

当我看完这本原版书后,感觉它写出了这几年我们正在研究、摸索而尚未能整理出来的许多东西,是从软件系统的角度对 ARM 系统开发的一个原则性指导,也是嵌入式软件开发者从入门变为高手的一个必备指南。作者论述简洁明了,把原本一些较复杂的问题(如嵌入式 OS),只用很少的篇幅就把一个基本框架说得很清楚,确有功底深厚,举重若轻之感。书中论述的一些知识、方法和技巧,也是做好一个嵌入式软件的基础。只有把许多细致的基础工作做扎实,才能聚沙成塔,编写出高效的软件,创造出具有竞争力的产品。翻译此书,希望能对目前国内 32 位(ARM)嵌入式系统教学、研发热潮中的广大专业技术人员有所帮助,并提升 32 位嵌入式系统应用的水平。

本书系统介绍了基于 ARM 的软件设计与优化方法,知识面较广,涉及了计算机学科的很多基础知识,包括 C 语言与汇编语言程序设计、编译原理、数据结构、操作系统、计算机体系结构、数值计算、数字信号处理等内容;同时也给翻译工作带来了一定的困难,许多词汇、术语若按字面上翻译,则很难确切地表达出其本意,我们主要根据其上下文内容和中文习惯,并参照其他相关的中文书籍,进行意译。如 5.9 节中的 aligned,译成“(地址)边界对齐的”,endianness 译成“字节排列方式(大/小端)”;6.2 节中的 profiler,译成“性能分析器”等。另外一些简单的常用词,如 load-store,在用作名词时不译,在用作动词时译成“装载-存储”,如“load 指令在装载数据时……”,而不译成“装载指令在装载数据时……”。关于 cache 写策略的 writethrough 和 writeback,本书中分别译为“直写”和“回写”,而没有用通常所说的“写直达”和“写回”,我们认为这样更简单明了而贴近原意。还有一些专业词汇,如 DSP 部分的“tap(抽头)”、“biquad(双阶节)”等,首次出现时都给出译文和英文,以后简单而都能理解的一般只给出英文,特别是用作单位时,如 cycles/tap 和 cycles/biquad 等。还有 ARMv5 和 ARMv6 体系结构的一些内容,几乎找不到相关资料可参考,也没有实验平台可以验证,只能根据我们的理解进行翻译。另外,对于一些程序中的简单注释,考虑到此书并非入门级读物,以及书中已有较详细的说明,经与出版社商量,没有全部翻译。

为了尊重原著,本书有些用词可能与国内一些书不太一致,例如原著中多处出现的“processor(处理器)”,是一种广义的泛指,可能是指一个 ARM 内核或内核中的一部分,也可能是指一个芯片;另外在一些地方(如表 2.10 中),把 ARM7TDMI, ARM9TDMI, ARM720T 及 ARM920T 等都统称为“CPU 核(core)”。所以本书也不区分“内核”与“核”。请读者阅读时注意。

在翻译过程中,发现了原著中的一些错误与不妥之处,通过与原作者多次交流并得到确认,在译稿中已得到纠正。还有一些标有“译者注”的地方,是我们根据对原文的理解,结合国内的习惯用法,作的一些补充说明。

本书对许多问题作了简明、细致的阐述,论述非常客观、公正。对一个事物的介绍,总是结合其背景或环境;对一个问题的描述或改进,一般也是从正、反两方面来进行,既说明了一种技术、方法对某方面的益处,同时也指出可能对另一方面造成的负面影响,即使对 ARM 处理器的介绍,也并非全是褒奖之词。这种严谨、科学的辩证思维和工作作风,更是值得我们每一个人学习并应身体力行的。

全书主要由笔者翻译并校对。我的研究生张群忠、沈颖、梁丹、庄艺唐、姜宁、吴红举及杨海波等参与了部分内容的翻译和资料整理工作。我的同事续晋华老师帮助修改和审核了第 8 章。我的同事杨艳琴老师通读了译稿,并提出了许多修改建议。原著的 3 位作者 Andrew N. Sloss, Dominic Symes 和 Chris Wright 及时提供了许多帮助,Andrew N. Sloss 先生还亲临我们实验室交流、指导。在此书的翻译过程中,还得到了 ARM(中国)总裁谭军博士、费浙平先生,北京航空航天大学出版社马广云博士、胡晓柏编辑,深圳英蓓特信息技术有限公司徐光峰先生,广州周立功单片机发展有限公司周立功先生的热情指导和鼓励。在此,谨向他(她)们表示衷心的感谢。

我也要特别感谢我的家人——妻子戴军、女儿沈维嘉,正是由于她们的无私帮助和全力支持,使我能全身心地投入到自己所热爱的工作之中,并在短短 6 个月的时间里完成了此书的翻译工作。

翻译是一件很难做得完美的事。由于时间仓促及水平所限,错误及不妥之处,请各位读者批评指正,并提出宝贵意见。我也会在我们实验室的网站(www.emlab.net)上及时发布相关信息,欢迎访问并相互交流。

沈建华

2005 年 2 月

于华东师范大学

前言

如今,嵌入式系统开发人员和片上系统设计人员越来越多地选择特定的处理器内核和配套的工具、库及现成的组件来快速开发基于微处理器的新产品。ARM 在这一方面表现尤为突出。在过去的 10 年中,ARM 体系结构已成为世界上最受欢迎的 32 位体系结构,在本书完成时,基于 ARM 的处理器已发售了超过 20 亿片。ARM 处理器已被嵌入到各种产品——从移动电话到汽车刹车系统。全球的 ARM 合作伙伴和第三方供应商,在半导体和产品设计公司中迅速发展壮大,包括硬件工程师、系统设计人员和软件开发人员。但至今还没有一本书能够较好地满足基于 ARM 的嵌入式系统及软件开发的需要,本书将填补这一空白。

本书的目标是,从一名产品开发者的角度来描述 ARM 内核的操作,重点放在软件设计上。由于本书是专门写给有一些嵌入式系统开发经验而可能对 ARM 体系结构不熟悉的工程师的,所以不要求有以前的 ARM 开发经验。

为了帮助读者尽快地学以致用,书中包含了一系列 ARM 软件范例。它们可以被集成到商业产品中,或者作为模板以快速创建应用软件。这些范例都已编号,读者可以在 Morgan Kaufmann 出版社的网站上方便地找到这些源代码。对于有 ARM 开发经验而想要获得 ARM 系统最高效能的人来说,这些代码同样颇有价值。

本书的结构

本书开头部分简要介绍了 ARM 处理器的设计原则,说明了它与传统 RISC 思想的区别及其原因。第 1 章还介绍了基于 ARM 处理器的简单嵌

ARM 嵌入式系统开发

入式系统。

第 2 章进一步深入到硬件,介绍了 ARM 处理器核,并综述了当前市场上的 ARM 内核。

第 3 章和第 4 章分别介绍了 ARM 和 Thumb 指令集,也为本书后面的内容打下基础;有一些关键指令的解释,包括完整的例子,因此这 2 章可以看作是指令集的使用指南。

通过我们在协助 ARM 客户工作时开发的许多例子,第 5 章和第 6 章讲述了如何编写高效的代码。第 5 章讲述了在 ARM 体系结构上编写可以被高效编译的 C 代码的技巧和规则。这些技巧和规则已得到了证实,并且有助于确定哪些代码应该被优化。第 6 章详述了编写和优化 ARM 汇编代码的最佳方法,这对于通过降低系统功耗和时钟频率来改善系统性能是至关重要的。

由于在许多算法中都用到一些基本操作,所以如何优化它们是很值得研究的。第 7 章讨论了如何针对 ARM 处理器优化基本操作。该章不仅提供了实现一般基本操作的优化参考,而且为想得到一种快速参考方法的使用者提供了更加复杂的数学运算的优化参考。对于想要深入研究各个实现的读者,还提供了所需的理论知识。

嵌入式音频和视频系统应用的需求正在日益增长。它们需要数字信号处理(DSP)能力,直到最近,这种处理能力一般仍是由独立的 DSP 芯片提供的。然而,现在 ARM 体系结构提供了更高的存储器带宽和快速乘累加运算,使得仅用一个 ARM 内核的设计就能支持这些应用。第 8 章研究了如何尽可能改善 ARM 在数字处理应用方面的性能以及怎样实现 DSP 算法。

异常处理是嵌入式系统的核心。高效的异常处理能够大大改善系统的性能。第 9 章以一系列翔实的范例介绍了异常处理和中断的理论与实践。

固件是所有嵌入式系统的重要部分。在第 10 章中,通过我们设计的、称为 Sandstone 的一个简单固件包,介绍了固件的结构和设计。本章还介绍了一些可以运行在 ARM 上的流行的、工业固件包。

第 11 章以我们设计的、称为简单小操作系统(SLOS)的一个简单嵌入式操作系统为例,示范了嵌入式操作系统的实现方法。

第 12,13,14 章重点关注存储系统。第 12 章讨论了围绕 ARM 内核的各种 cache 技术,演示了带 cache 的特定 ARM 处理器的 cache 控制例程。第 13 章讨论了存储器保护单元(MPU)。第 14 章讨论了存储器管理单元(MMU)。

最后,在第 15 章,展望了 ARM 体系结构的未来,重点讲述今后几年内 ARM 指令集的发展方向和正在实现的新技术。

附录提供了详细的指令集参考、周期定时以及特定的 ARM 产品。

网上的范例

如前所述,本书有大量的经过测试的实用程序,可以帮助强化有关概念和方法。它们在 Morgan Kaufmann 出版社的网站上 <http://www.mkp.com/companions/1558608745> 可以找到。

致 谢

首先要感谢的,当然是我们的妻子——Shau Chin Symes 和 Yulian Yang,以及所有大力支持并容忍我们花费了大量家庭生活时间在这一项目上的亲人。

本书前后花了许多年,很多人都曾给予过鼓励和技术上的建议,我们要感谢所有这些人。写一本技术书需要很辛苦地重视大量细节问题,因此要感谢所有投入时间和精力阅读本书并给予反馈的审稿人员。在这个过程中,与出版商一起工作的审稿人员有 Jim Turley (Silicon - Insider); Peter Maloy (Code Sprite); Chris Larsen, Peter Harrod (ARM, Ltd.); Gary Thomas (MLB Associate); Wayne Wolf (Princeton University); Scott Runner (Qualcomm, Inc.); Nial Murphy (PanelSoft) 和 Dominic Sweetman (Algorithmics, Ltd.)。

要特别感谢 Wilco Dijkstra, Edward Nevill 和 David Seal,允许我们在本书中使用一些精选的范例。还要感谢 Rod Crawford, Andrew Cummins, Dave Flynn, Jamie Smith, William Ree 和 Anne Rooney,在整个过程中提供了帮助和建议。感谢 ARM 战略支持小组: Howard Ho, John Archibald, Miguel Echavarria, Robert Allen 和 Ian Field,阅读并提供了快速反馈。

我们还要感谢 John Rayfield 发起这个项目并完成了第 15 章。还要感谢 David Brash 审阅了原稿,并允许我们在本书中使用了 ARMv6 的一些资料。

最后,我们要感谢 Morgan Kaufmann 出版社,特别是 Denise Penrose 和 Belinda Breyer 在整个项目过程中的耐心和建议。

目 录

第 1 章 基于 ARM 的嵌入式系统	1	2.4 异常、中断及向量表	27
1.1 RISC 设计思想	2	2.5 内核扩展	28
1.2 ARM 设计思想	3	2.5.1 cache 和紧耦合存储器	29
1.3 嵌入式系统的硬件	5	2.5.2 存储管理	30
1.3.1 ARM 总线技术	6	2.5.3 协处理器	31
1.3.2 AMBA 总线协议	6	2.6 体系结构的不同版本	31
1.3.3 存储器	7	2.6.1 命名规则	32
1.3.4 外 设	9	2.6.2 体系结构的发展	33
1.4 嵌入式系统的软件	10	2.7 ARM 处理器系列	34
1.4.1 初始化(启动)代码	10	2.7.1 ARM7 系列	35
1.4.2 操作系统	11	2.7.2 ARM9 系列	36
1.4.3 应用程序	12	2.7.3 ARM10 系列	37
1.5 总 结	12	2.7.4 ARM11 系列	37
第 2 章 ARM 处理器基础	14	2.7.5 专用处理器	37
2.1 寄存器	16	2.8 总 结	38
2.2 当前程序状态寄存器	17	第 3 章 ARM 指令集	39
2.2.1 处理器模式	18	3.1 数据处理指令	42
2.2.2 分组寄存器	18	3.1.1 MOVE 指令	42
2.2.3 状态和指令集	21	3.1.2 桶形移位器	43
2.2.4 中断屏蔽	22	3.1.3 算术指令	46
2.2.5 条件标志	22	3.1.4 算术指令使用桶形移位器	47
2.2.6 条件执行	24	3.1.5 逻辑指令	48
2.3 流水线	24		

ARM 嵌入式系统开发

3.1.6 比较指令.....	49	5.2.1 局部变量类型.....	97
3.1.7 乘法指令.....	50	5.2.2 函数参数类型	101
3.2 分支指令.....	51	5.2.3 有符号数与无符号数	103
3.3 load-store 指令	53	5.3 C 循环结构	104
3.3.1 单寄存器传送指令.....	53	5.3.1 固定次数的循环	104
3.3.2 单寄存器 load-store 指令的 寻址方式.....	54	5.3.2 不定次数的循环	107
3.3.3 多寄存器传送指令.....	57	5.3.3 循环展开	108
3.3.4 交换指令.....	65	5.4 寄存器分配	111
3.4 软件中断指令.....	66	5.5 函数调用	113
3.5 程序状态寄存器指令.....	68	5.6 指针别名	117
3.5.1 协处理器指令.....	69	5.7 结构体安排	120
3.5.2 协处理器 15(CP15)指令语法	70	5.8 位 域	124
3.6 常量的装载.....	71	5.9 边界不对齐数据和字节排列方式 (大/小端).....	127
3.7 ARMv5E 扩展	72	5.10 除 法.....	131
3.7.1 零计数指令.....	73	5.10.1 带余数的无符号重复除法	133
3.7.2 饱和算术指令.....	73	5.10.2 把除转换为乘.....	133
3.7.3 ARMv5E 乘法指令	74	5.10.3 除数是常数的无符号除法	136
3.8 条件执行.....	75	5.10.4 除数是常数的有符号除法	137
3.9 总 结.....	77	5.11 浮点运算.....	140
第 4 章 Thumb 指令集	78	5.12 内联函数和内嵌汇编.....	140
4.1 Thumb 寄存器的使用	81	5.13 移植问题.....	144
4.2 ARM-Thumb 交互	82	5.14 总 结.....	145
4.3 其它分支指令.....	84	第 6 章 ARM 汇编与优化	147
4.4 数据处理指令.....	84	6.1 编写汇编代码	148
4.5 单寄存器 load-store 指令	87	6.2 性能分析和周期计数	154
4.6 多寄存器 load-store 指令	89	6.3 指令调整	154
4.7 堆栈指令.....	90	6.4 寄存器分配	162
4.8 软件中断指令.....	91	6.4.1 分配变量给寄存器	163
4.9 总 结.....	92	6.4.2 使用超过 14 个的局部变量	166
第 5 章 高效的 C 编程	93		
5.1 C 编译器及其优化概述.....	94		
5.2 基本的 C 数据类型	96		

6.4.3 最大限度地使用寄存器	168	7.3 除 法	208
6.5 条件执行	172	7.3.1 通过试探减法实现无符号 数除法	208
6.6 循环结构	174	7.3.2 无符号整数的 Newton- Raphson 除法	215
6.6.1 减计数循环	174	7.3.3 无符号小数 Newton- Raphson 除法	221
6.6.2 展开计数循环	175	7.3.4 有符号数除法	228
6.6.3 多层嵌套循环	178	7.4 平方根	229
6.6.4 其它计数循环	181	7.4.1 通过试探减法计算平方根	229
6.7 位操作	182	7.4.2 使用 Newton-Raphson 迭 代计算平方根	231
6.7.1 固定宽度的位域打包和解包	182	7.5 超越函数: log, exp, sin, cos	233
6.7.2 可变宽度编码的位流打包	183	7.5.1 以 2 为底的对数运算	233
6.7.3 可变宽度编码的位流解包	186	7.5.2 2 的乘幂	235
6.8 高效的 switch	188	7.5.3 三角函数	236
6.8.1 在范围 $0 \leq x < N$ 的 switch	189	7.6 字节顺序反转和位操作	239
6.8.2 基于通用变量 x 的 switch	191	7.6.1 字节顺序反转	240
6.9 边界不对齐数据的处理	192	7.6.2 位变换	240
6.10 总 结	196	7.6.3 '1' 位计数	243
第 7 章 基本运算优化	197	7.7 饱和及舍入运算	244
7.1 双精度整数乘法	198	7.7.1 饱和 32 位数到 16 位	245
7.1.1 长整型乘法	199	7.7.2 饱和左移	245
7.1.2 128 位结果的无符号 64 位 乘法	200	7.7.3 舍入右移	245
7.1.3 128 位结果的有符号 64 位整数 乘法	201	7.7.4 饱和的 32 位加减法	245
7.2 整数规格化和前导 0 计数	203	7.7.5 饱和绝对值	246
7.2.1 ARMv5 及以上体系结构 的整数规格化	203	7.8 随机数产生	246
7.2.2 在 ARMv4 体系结构上的 规格化	204	7.9 总 结	247
7.2.3 后缀 0 计数	206	第 8 章 数字信号处理	248
		8.1 表示一个数字信号	250
		8.1.1 选择一种表示方法	250
		8.1.2 操作以定点格式存储的值	253

ARM 嵌入式系统开发

8.1.3	定点信号的加法和减法	254	9.3.5	优先级标准中断处理	340
8.1.4	定点信号的乘法	255	9.3.6	优先级直接中断处理	344
8.1.5	定点信号的除法	256	9.3.7	优先级分组中断处理	347
8.1.6	定点信号的平方根	256	9.3.8	基于 VIC PL190 的中断服务 例程	351
8.1.7	小结:数字信号的表示	256	9.4	总 结	352
8.2	基于 ARM 的 DSP 介绍	258	第 10 章	固 件	353
8.2.1	ARM7TDMI 的 DSP	259	10.1	固件和引导装载程序	354
8.2.2	ARM9TDMI 的 DSP	260	10.1.1	ARM Firmware Suite	357
8.2.3	StrongARM 的 DSP	262	10.1.2	Red Hat Redboot	358
8.2.4	ARM9E 的 DSP	264	10.2	例子:Sandstone	358
8.2.5	ARM10E 的 DSP	265	10.2.1	Sandstone 的目录结构	359
8.2.6	Intel Xscale 的 DSP	267	10.2.2	Sandstone 的代码结构	359
8.3	FIR 滤波器	269	10.3	总 结	364
8.4	IIR 滤波	284	第 11 章	嵌入式操作系统	365
8.5	离散傅里叶变换	292	11.1	基本模块	366
8.6	总 结	304	11.2	实例:简单小型操作系统 SLOS	367
第 9 章	异常和中断处理	306	11.2.1	SLOS 目录结构	368
9.1	异常处理	307	11.2.2	初始化	369
9.1.1	ARM 处理器模式及异常	307	11.2.3	存储模型	373
9.1.2	向量表	309	11.2.4	中断和异常处理	374
9.1.3	异常优先级	310	11.2.5	调度程序	378
9.1.4	链接寄存器偏移	311	11.2.6	上下文切换	380
9.2	中 断	313	11.2.7	设备驱动程序框架	382
9.2.1	分配中断	313	11.3	总 结	383
9.2.2	中断延迟	314	第 12 章	高速缓冲存储器 cache	385
9.2.3	IRQ 与 FIQ 异常	315	12.1	存储层次和 cache	387
9.2.4	基本的中断堆栈设计与实现	317	12.2	cache 结构	390
9.3	中断处理方法	321	12.2.1	cache 存储器的基本结构	391
9.3.1	非嵌套中断处理	321			
9.3.2	嵌套中断处理	324			
9.3.3	可重入中断处理	330			
9.3.4	优先级简单中断处理	334			

12.2.2	cache 控制器的基本操作	392	12.6.4	在 Intel XScale SA-110 中 锁定 cache 行	437
12.2.3	cache 与主存的关系	392	12.7	cache 与软件性能	440
12.2.4	组相联	395	12.8	总 结	441
12.2.5	写缓冲器	399	第 13 章	存储器保护单元 MPU	444
12.2.6	cache 效率的衡量	399	13.1	受保护的区域	446
12.3	cache 策略	400	13.1.1	重叠区域	447
12.3.1	写策略——直写法或回写法	400	13.1.2	背景区域	448
12.3.2	cache 行替换策略	401	13.2	初始化 MPU, cache 和写缓冲器	449
12.3.3	cache 失效时的分配策略	404	13.2.1	定义区域的大小和位置	450
12.4	协处理器 15 与 cache	405	13.2.2	访问权限	453
12.5	清除和清理 cache	406	13.2.3	设置区域的 cache 和写缓冲器 属性	457
12.5.1	清除 cache	407	13.2.4	使能区域和 MPU	460
12.5.2	清理 cache	410	13.3	MPU 系统示例	461
12.5.3	清理 D-cache	410	13.3.1	系统需求	462
12.5.4	使用路和组索引寻址清理 D-cache	414	13.3.2	使用存储器映射分配区域	463
12.5.5	使用 test-clean 命令清理 D-cache	417	13.3.3	初始化 MPU	464
12.5.6	在 Intel XScale SA-110 和 Intel StrongARM 内核中 清理 D-cache	418	13.3.4	初始化和配置区域	465
12.5.7	清理和清除部分 cache	421	13.3.5	完成初始化 MPU	468
12.6	cache 锁定	426	13.3.6	受保护系统的上下文切换	469
12.6.1	在 cache 中锁定代码和数据	427	13.3.7	mpuSLOS	470
12.6.2	通过增加路索引来锁定 cache	428	13.4	总 结	470
12.6.3	使用锁定位锁定 cache	433	第 14 章	存储管理单元	472
			14.1	从 MPU 到 MMU	474
			14.2	虚存如何工作	474
			14.2.1	使用页定义区域	476
			14.2.2	多任务和 MMU	478
			14.2.3	虚存系统的存储器组织	480

ARM 嵌入式系统开发

14.3	ARM MMU 的详情	482
14.4	页 表	482
14.4.1	一级页表项	483
14.4.2	L1 转换表基地址	484
14.4.3	二级页表项	485
14.4.4	为嵌入式系统选择合适的 页大小	486
14.5	转换旁路缓冲器	487
14.5.1	单步页表搜索	487
14.5.2	2 步页表搜索	488
14.5.3	TLB 操作	489
14.5.4	TLB 锁定	490
14.6	域和存储器访问权限	491
14.7	cache 和写缓冲器	493
14.8	协处理器 CP15 和 MMU 配置	494
14.9	快速上下文切换扩展	496
14.9.1	FCSE 如何使用页表和域	497
14.9.2	使用 FCSE 的提示	499
14.10	示例:一个简单的虚拟存储系统	500
14.10.1	第 1 步:定义固定的系统 软件区域	501
14.10.2	第 2 步:为每个任务定义 虚存映射	502
14.10.3	第 3 步:在物理存储器中 定位区域	503
14.10.4	第 4 步:定义和定位页表	503
14.10.5	第 5 步:定义页表和区域 数据结构	506
14.10.6	第 6 步:初始化 MMU、Cache 和写缓冲器	509

14.10.7	第 7 步:建立上下文切换程序	524
14.11	MMUSLOS 示例	525
14.12	总 结	525
第 15 章	ARM 体系结构的发展	527
15.1	ARMv6 对高级 DSP 和 SIMD 的支持	528
15.1.1	SIMD 算法操作	529
15.1.2	打包指令	532
15.1.3	复数运算支持	533
15.1.4	饱和指令	534
15.1.5	绝对差值求和指令	534
15.1.6	双 16 位乘法指令	535
15.1.7	高位字乘法	536
15.1.8	密码算法乘法扩展	537
15.2	ARMv6 增加的系统和多处理 器支持	538
15.2.1	混合大小端支持	538
15.2.2	异常处理	538
15.2.3	多处理同步原语(Multipro- cessing Synchronization Primitives)	540
15.3	ARMv6 的实现	541
15.4	ARMv6 之后的未来技术	543
15.4.1	TrustZone	543
15.4.2	Thumb-2	543
15.5	总 结	544
附录 A	ARM 和 Thumb 汇编指令	546
A.1	如何使用这篇附录	547
A.2	语 法	548
A.2.1	可选表达式	548
A.2.2	寄存器	548
A.2.3	立即数	548
A.2.4	条件和标志	549

A.2.5 移位操作	550	D.1 指令周期定时表的使用	627
A.3 按字母顺序列出 ARM 和 Thumb 指令	551	D.2 ARM7TDMI 指令周期定时	628
A.4 ARM 汇编速查	597	D.3 ARM9TDMI 指令周期定时	630
A.4.1 ARM 汇编变量	598	D.4 StrongARM1 指令周期定时	631
A.4.2 ARM 汇编标注	599	D.5 ARM9E 指令周期定时	632
A.4.3 ARM 汇编表达式	600	D.6 ARM10E 指令周期定时	634
A.4.4 ARM 汇编保留字	601	D.7 Intel XScale 指令周期定时 ...	636
A.5 GNU 汇编快速查询	608	D.8 ARM11 指令周期定时	637
附录 B ARM 和 Thumb 指令编码	612	附录 E 建议的参考读物	642
B.1 ARM 指令集编码	613	E.1 ARM 参考	643
B.2 Thumb 指令集编码	618	E.2 算法参考	643
B.3 程序状态寄存器	620	E.3 存储器管理与 cache 体系结构 (硬件综述与参考)	643
附录 C 处理器与体系结构	622	E.4 操作系统参考	644
C.1 ARM 命名规则	623		
C.2 内核与体系结构	624		
附录 D 指令周期定时	626		

第 1 章

基于 ARM 的嵌入式系统

- RISC 设计思想
- ARM 设计思想
- 嵌入式系统的硬件
- 嵌入式系统的软件
- 总 结

ARM 嵌入式系统开发

在许多成功的 32 位嵌入式系统中,ARM 处理器都是其核心组成部分。也许大家还没有意识到,自己身边可能就有基于 ARM 处理器的产品。ARM 内核已被广泛应用于移动电话、掌上设备以及种类繁多的便携式消费类产品中。

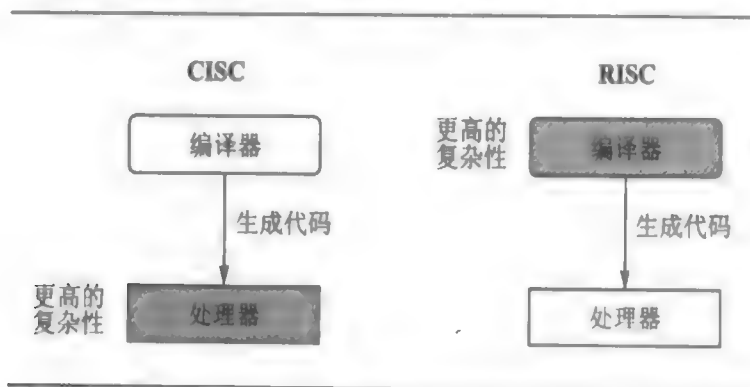
从 1985 年第一个 ARM1 原型诞生至今,ARM 的设计者们经历了漫长的探索之路。到 2001 年底,已经有超过 10 亿个 ARM 处理器被销售到了世界各地。ARM 公司的成功建立在一个简单而又强大的原始设计之上,随着技术的不断创新,这个设计也在不断改进。事实上,ARM 内核并不是单一的,而是一个遵循相同设计理念和使用相同指令集的内核系列。

例如,最成功的 ARM 内核之一 ARM7TDMI,具有最高 120 Dhrystone MIPS(注:Dhrystone MIPS version 2.1 是一个小的基准测试程序)的性能、高的代码密度和低功耗等特性,使它成为移动嵌入式设备的最佳选择。

在第 1 章中,首先介绍 ARM 是如何采用 RISC(精简指令集计算机)的设计思想,创造出一个灵活的处理器结构的;然后通过介绍一个嵌入式设备的实例,讨论围绕 ARM 处理器的典型的软硬件相关技术。

1.1 RISC 设计思想

ARM 内核采用 RISC 体系结构。RISC 是一种设计思想,其目标是设计出一套能在高时钟频率下单周期执行,简单而有效的指令集。RISC 的设计重点在于降低由硬件执行的指令的复杂度,这是因为软件比硬件容易提供更大的灵活性和更高的智能。因此,RISC 设计对编译器有更高的要求;相反,传统的复杂指令集的计算机(CISC)则更侧重于硬件执行指令的功能性,使 CISC 指令变得更复杂。图 1.1 比较了两者的主要不同。



CISC 强调硬件的复杂性;RISC 注重编译器的复杂性

图 1.1 CISC 对 RISC

RISC 设计思想主要由下面 4 个设计准则来实现。

- **指令集**——RISC 处理器减少了指令种类。RISC 的指令种类只提供简单的操作,使一个周期就可以执行一条指令。编译器或者程序员通过几条简单指令的组合来实现一个复杂的操作(例如,除法操作)。每条指令的长度都是固定的,允许流水线在当前指令译码阶段去取其下一条指令;而在 CISC 处理器中,指令的长度通常不固定,执行也需要多个周期。
- **流水线**——指令的处理过程被拆分成几个更小的、能够被流水线并行执行的单元。在理想情况下,流水线每周期前进一步,可获得最高的吞吐率;而 CISC 指令的执行需要调用微代码的一个微程序。
- **寄存器**——RISC 处理器拥有更多的通用寄存器。每个寄存器都可存放数据或地址。寄存器可为所有的数据操作提供快速的局部存储访问;而 CISC 处理器都是用于特定目的的专用寄存器。
- **load-store 结构**——处理器只处理寄存器中的数据。独立的 load 和 store 指令用来完成数据在寄存器和外部存储器之间的传送。因为访问存储器很耗时,所以把存储器访问和数据处理分开。这样有一个好处,那就是可反复地使用保存在寄存器中的数据,而避免多次访问存储器。相反,在 CISC 结构中,处理器能够直接处理存储器中的数据。

这些设计准则,使得 RISC 结构的处理器更为简单,因此内核能够工作在更高时钟频率。相反,传统的 CISC 处理器因为结构更为复杂,只能工作在较低的时钟频率。然而,经过 20 多年的发展,CISC 处理器也引入了许多 RISC 的设计思想,RISC 和 CISC 之间的界线已经变得越来越模糊了。

1.2 ARM 设计思想

有许多客观需求促进了 ARM 处理器的设计改进。首先,便携式的嵌入式系统往往需要电池供电。为降低功耗,ARM 处理器已被特殊设计成较小的核,从而延长了电池的使用时间。特别是对于某些应用,如移动电话和个人数字助理(PDAs)等,这一点是非常重要的。

高的代码密度是嵌入式系统的又一个重要需求。由于成本问题和物理尺寸的限制,嵌入式系统的存储器是很有限的。所以,高的代码密度对于那些只限于在板存储器的应用是非常有帮助的,例如移动电话和海量存储设备等。

另外,嵌入式系统通常都是价格敏感的,因此一般都使用速度不高,成本较低的存储器。对于数码相机之类的大宗产品,在设计时每一分成本都需要考虑。因此,若处理器能使用低成本的存储器,将大大降低系统成本。

另外一个重要的需求就是要缩小嵌入式处理器内核管芯(die)的面积。对于一个单片方案,处理器内核所占用的面积越小,留给外设电路的空间就越大。这可以减少最终产品的外围芯片数目,从而降低设计和制造成本。

ARM 处理器中已经集成了硬件调试技术,因此软件工程师们能够观察到处理器在执行代码时的具体情况。这种更高的可视性,使得软件工程师们能够更快速地解决问题。这对缩短产品上市时间有直接的效果,从而降低了系统的整体开发成本。

ARM 内核不是一个纯粹的 RISC 体系结构,这是为了使它能够更好地适应其主要应用领域——嵌入式系统。在某种意义上,甚至可以认为 ARM 内核的成功,正是因为它没有在 RISC 概念上沉入太深。现在系统的关键并不在于单纯的处理器速度,而在于有效的系统性能和功耗。

面向嵌入式系统的指令集

为了使 ARM 指令集能够更好地满足嵌入式应用的需要,ARM 指令集和单纯的 RISC 定义有以下几方面的不同。

- **一些特定指令的周期数可变**——并不是所有的 ARM 指令都是单周期的。例如:多寄存器装载/存储的 load/store 指令的执行周期就是不确定的,须根据被传送的寄存器个数而定。如果是访问连续的存储器地址,就可以改善性能,因为连续的内存访问通常比随机访问要快。同时,代码密度也得到了提高,因为在函数的起始和结尾,多个寄存器的传输是很常用的操作。
- **内嵌桶形移位器产生了更为复杂的指令**——内嵌桶形移位器是一个硬件部件,在一个输入寄存器被一条指令使用之前,内嵌桶形移位器可以处理该寄存器中的数据。它扩展了许多指令的功能,以此改善了内核的性能,提高了代码密度。我们将在第 2~4 章中详细讲解这一特性。
- **Thumb 16 位指令集**——ARM 内核增加了一套称之为 Thumb 指令的 16 位指令集,使得内核既能够执行 16 位指令,也能够执行 32 位指令,从而增强了 ARM 内核的功能。16 位指令与 32 位的定长指令相比较,代码密度可以提高约 30%。
- **条件执行**——只有当某个特定条件满足时指令才会被执行。这个特性可以减少分支指令的数目,从而改善了性能,提高了代码密度。
- **增强指令**——一些功能强大的数字信号处理器(DSP)指令被加入到标准的 ARM 指令之中,以支持快速的 16×16 位乘法操作及饱和运算。在某些应用中,传统的方法需要微处理器加上 DSP 才能实现。ARM 的这些增强指令,使得 ARM 处理器也能够满足这些应用的需要。

这些增加的特性使得 ARM 处理器成为当今最通用的 32 位嵌入式处理器内核之一。

许多世界著名的半导体公司都在制造基于 ARM 内核的产品。

1.3 嵌入式系统的硬件

嵌入式系统可控制各种不同的设备,从生产线上的小传感器,到 NASA 太空探测器中的实时控制系统。所有的这些设备都是一些软件和硬件部件的组合。每一个部件的选择都要考虑到效率;而且如果可行,设计时还需要考虑将来的扩展性和延伸性。

图 1.2 所示为一个典型的使用 ARM 内核的嵌入式器件。每一个方框表示了一个功能或特性。方框之间的连线是传送数据的总线。

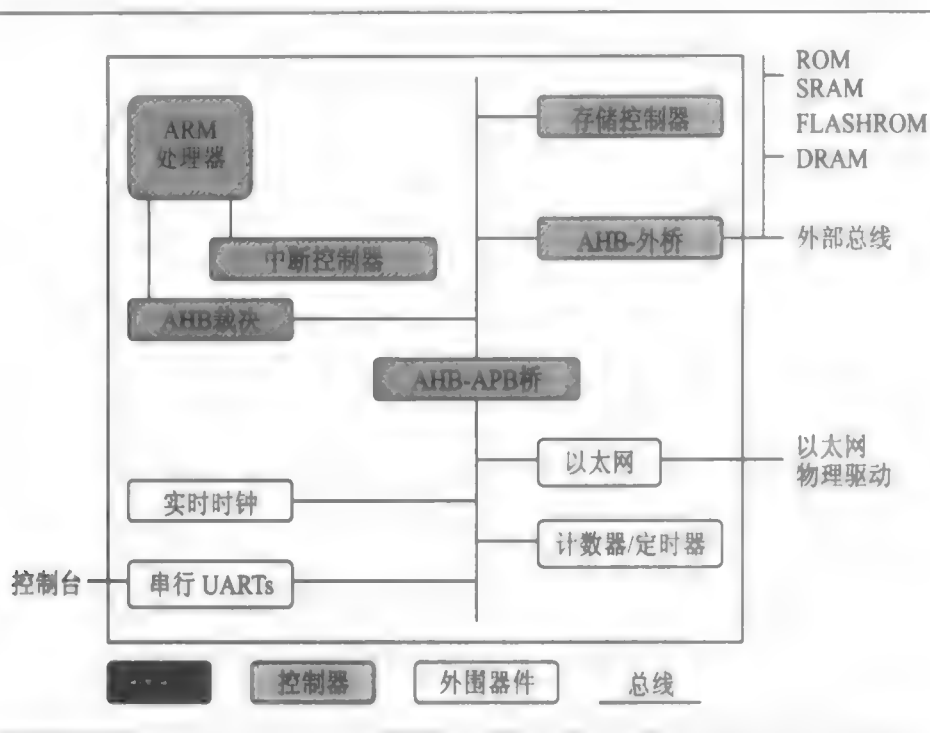


图 1.2 一个微控制器——基于 ARM 的嵌入式器件

可以把这个器件分为以下 4 个主要的硬件部分。

- **ARM 处理器**——控制整个器件。有多种版本的 ARM 处理器,以满足不同的处理特性。一个 ARM 处理器包含了一个内核(执行引擎,用来执行指令和操纵数据)以及一些外围部件,它们之间由总线连接。这些部件可能包括存储器管理和 cache。
- **控制器**——协调系统的重要功能模块。两个常见的控制器是中断控制器和存储器控制器。
- **外设**——提供芯片与外部的所有输入/输出功能,器件之间的一些独有特性就是靠不同的外设来体现的。

- 总线——用于在器件不同部件之间进行通信。

1.3.1 ARM 总线技术

嵌入式系统使用和 x86 PC 不同的总线技术。PCI(Peripheral Component Interconnect)总线技术是一种最常见的 PC 总线技术,用于把诸如图形显示适配器和硬盘控制器等连接到 x86 处理器总线。这是一种 PC 机主板上的外部或片外总线(这种总线也定义了外部设备与系统连接的机械和电气特性)。

相反,嵌入式设备使用芯片内的片上总线,允许不同的外围器件能够和 ARM 内核互联。

有两种不同类型的逻辑设备连接到总线:ARM 处理器是总线主设备(master)——拥有对总线的仲裁权,通过同一总线,该逻辑设备可主动发起数据传输请求;外围器件是总线从设备(slave)——在总线上是被动的,逻辑设备只能对主设备发出的一个传输请求作出反应。

总线可分为两个结构层:第一层是物理层,定义一些电气特征和总线宽度(16,32 或 64 位);第二层是协议层(protocol),定义了处理器和外围设备之间进行数据通信的逻辑规则。

ARM 主要是一个芯片设计公司,它几乎不实现具体的总线电气特性,但它详细地定义了总线协议。

1.3.2 AMBA 总线协议

高级微控制总线结构(AMBA)于 1996 年被提出,并被 ARM 处理器广泛用作片上总线结构。最初的 AMBA 总线包含 ARM 系统总线(ASB)和 ARM 外设总线(APB)。之后,ARM 公司提出了另一种总线设计,称为 ARM 高性能总线(AHB)。使用 AMBA,外设设计者可在多个项目中重复使用同一设计。由于已有大量的基于 AMBA 接口设计的外设,因此硬件芯片设计者可从已做过测试和验证的外设中做广泛的选择。同样,一个外设也可以被简单地连接到片上总线,而不需要为不同的处理器结构重新设计接口。这种即插即用的接口提高了硬件开发者的设计效率,并缩短了产品上市时间。

AHB 能够提供比 ASB 更高的数据吞吐率,因为它不同于 ASB 的双向总线设计。AHB 是基于集中多总线机制(centralized multiplexed bus scheme)的。这种改变使得 AHB 总线能够在更高的时钟速度下运行,并成为第一个支持 64 位和 128 位宽度的 ARM 总线。ARM 的 AHB 总线结构还引入了两个变体:多层 AHB 和 AHB-Lite。原来的 AHB 在任何时候都只允许一个主设备活动在总线上,而多层 AHB 允许多个活动的总线主设备。AHB-Lite 是标准 AHB 总线的一个子集,也只允许一个总线主设备。这种总线是针对那些不需

要标准 AHB 全部特性的情况而设计的。

标准 AHB 和多层 AHB 的主从协议是相同的,但有不同的互联。多层 AHB 新的互联有利于多处理器系统,它允许并发操作和更高的吞吐率。

在图 1.2 所示例子的器件中有 3 条总线:一条 AHB 总线连接高性能的片内外设;一条 APB 总线连接较慢的片内外设;第 3 条总线用于连接片外的外设,这条总线不一定都存在,在这个例子中是有的,这条外部总线需要一个特殊的桥,用于和 AHB 总线连接。

1.3.3 存储器

一个嵌入式系统必须有一定的存储器来存放和执行代码。在决定存储器的层次、宽度和类型等特性时,必须综合考虑价格、性能和功耗等因素。如果为了获得所需要的带宽,存储器的运行速度必须提高 1 倍,那么功耗也会提高。

1.3.3.1 存储层次

几乎所有的计算机系统的存储器结构都是分层的。图 1.2 显示的是一个支持片外存储器的器件。处理器内部有一个可选的 cache(在图 1.2 中没有画出),它可以提高存储器的性能。

图 1.3 显示了一些存储器指标之间的关系(trade-offs):最快的存储器 cache,其物理位置上最靠近 ARM 处理器核;而最慢的辅助存储器则处在较远的位置。通常越靠近处理器核的存储器就越昂贵,容量也越小。

cache 位于主存和内核之间,用于提高处理器和主存之间的数据传输速度。cache 改善了系统的总体性能,但也使代码的执行时间变得不可预测,对实时系统响应也没有什么帮助。因此在许多小型的嵌入式系统中,并不需要使用 cache 来优化性能。

主存的容量较大——256 KB~256 MB (甚至更大),根据具体的应用而定,通常是一些独立的芯片。load/store 指令存取数据时,如果数据在 cache 中,则快速访问 cache;否则就访问主存。在各级存储器中,辅助存储器容量最大,但速度最慢,例如硬盘和 CD-ROM 等存储设备。如今的辅助存储器的容量可以从 600 MB 到上百 GB。

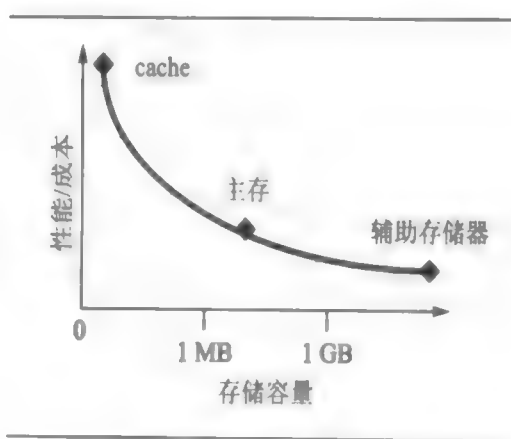


图 1.3 存储器指标关系图(trade-offs)

1.3.3.2 存储器数据宽度

存储器的数据宽度是指每次访问所返回的数据位数。典型的有 8, 16, 32 和 64 位。存储器宽度对系统整体的性价比会有直接的影响。

如果一个没有 cache 的系统使用 32 位 ARM 指令和 16 位宽度的存储器芯片, 则处理器每次取指就需要 2 个 16 位的存储器访问, 这显然会降低系统的性能, 但 16 位宽度的存储器价格会相对便宜。

与之相比, 如果内核执行 16 位的 Thumb 指令, 则对于 16 位宽度的存储器将获得更好的性能, 因为处理器获取每条指令只需一次存储器访问。因此, 对于 16 位宽度的存储器, 使用 Thumb 指令可获得性能和成本两方面的优势。

表 1.1 总结了当使用不同宽度的存储器时, 理论上获取一条指令所需要的周期数。

表 1.1 从存储器取指

指令长度	8 位存储器	16 位存储器	32 位存储器
ARM 32 位	4 周期	2 周期	1 周期
Thumb 16 位	2 周期	1 周期	1 周期

1.3.3.3 存储器类型

存储器的类型很多, 在本小节中, 我们将介绍一些基于 ARM 的嵌入式系统较常用的存储器。

- **只读存储器 ROM (Read-Only Memory)** 在所有类型的存储器中, ROM 是最不灵活的一种, 因为它里面的内容是在生产时就固定的, 不可再次编程来改变。ROM 常应用于不需要更新和修改内容的大宗产品, 也有许多设备使用 ROM 来存放启动代码。
- **Flash ROM (闪存)** 既可以读, 也可以写。但是它写的速度较慢, 因此不适合存放动态数据。它主要用于存放设备固件 (firmware) 和断电后仍需长期保存的数据。对 Flash ROM 的擦除和改写是完全由软件实现的, 不需要任何的硬件电路, 这样降低了制造成本。Flash ROM 已经成为当前最流行的只读存储器, 可选择 Flash ROM 用于满足对存储器的大容量需求或用于构建辅助存储器。
- **动态随机访问存储器 (DRAM)** 是设备中最为常用的 RAM。和其它类型的 RAM 相比, 它每兆字节 (MB) 的价格最低。DRAM 动态地——需要定时地 (ms 级) 刷新存储单元, 因此在使用 DRAM 前, 先要设置好 DRAM 控制器。
- **静态随机访问存储器 (SRAM)** 比传统的 DRAM 要快, 但它需要更大的硅片面积。SRAM 是静态的——不需要刷新。SRAM 的存取时间比 DRAM 要短得多, 因为

SRAM 在数据访问之间不需要暂停。由于它价格较高,因此通常用于容量小、速度快的情况,如高速存储器和 Cache。

- **同步动态随机访问存储器(SDRAM)** 是众多 DRAM 种类中的一种。它能够工作在比普通存储器更高的时钟频率下。因为 SDRAM 使用时钟,所以它和处理器总线是同步的。数据从存储元(memory cell)被流水化地取出,最后突发式(burst)地输出到总线。以前老的 DRAM 是异步的,不能像 SDRAM 这样突发式高效传输。

1.3.4 外 设

嵌入式系统和外界交互需要一定形式的外设。外设通过和片外其它设备或传感器的连接来实现芯片的输入/输出功能。每一个外设通常都只有单一的功能,也可以内置在芯片上。外设种类很多,可从一个简单的串行通信设备到非常复杂的 802.11 无线设备。

所有的 ARM 外设都是存储器映射的。编程接口是一组对应于某些存储器地址的寄存器。这些寄存器的地址是从某个特定外设的基地址开始的偏移量。

控制器是特殊的外设,可在一个嵌入式系统中实现更高层的功能。两个重要类型的控制器就是存储器控制器和中断控制器。

1.3.4.1 存储器控制器

各种不同类型的存储器通过存储器控制器连接到处理器总线上。上电时,存储器控制器由硬件配置,使得某些存储器处于工作状态。这些存储器允许执行初始化代码。有些存储器必须通过软件设置才能使用,比如在使用 DRAM 之前,必须首先设置存储器定时和刷新率。

1.3.4.2 中断控制器

当一个外设或器件需要服务时,它就向处理器提出一个中断请求。中断控制器提供一套可编程的管理机制,使软件通过设置中断控制寄存器中的相应位,来决定在任何特定时刻,哪一个外设或器件可以中断处理器。

ARM 处理器有两种中断控制器:标准的中断控制器和向量中断寄存器(VIC)。

标准中断控制器在一个外部设备需要服务时,发送一个中断请求信号给处理器核。控制器可以通过编程设置来忽略或屏蔽某个或某些设备的中断请求。中断处理程序读取在中断控制器中与各设备对应的表示中断请求的寄存器内容,从而判断哪个设备需要服务。

VIC 比标准中断控制器的功能更为强大,因为它区分中断的优先级,简化了判断中断源的过程。每个中断都有相应的优先级和处理程序地址,只有当一个新的中断优先级高于当前正在执行的中断处理优先级时,VIC 才向内核提出中断请求。根据类型的不同,VIC 可以调用标准的中断异常处理程序。该程序能够从 VIC 中读取设备的处理程序的地址,也

可使内核直接跳转到设备的处理程序执行。

1.4 嵌入式系统的软件

一个嵌入式系统需要软件来实现具体的应用。图 1.4 中所示的 4 个软件部分是嵌入式设备软件的典型构成。从底层的硬件设备层往上,每个软件层次逐层封装代码,使代码与硬件设备分离。

初始化代码是在板上最先被执行的代码,是针对一个或一组特定的目标而定制的。它在把控制权交给操作系统之前,初始化板上的最基本部分设备。

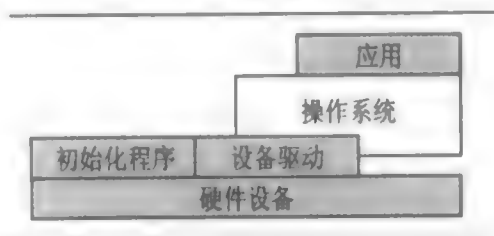


图 1.4 在硬件上执行的软件层次

操作系统提供了一个基础框架来控制应用程序和管理硬件系统资源。许多嵌入式系统都不需要一个完整的操作系统,而只需要一个由事件或轮询驱动的简单任务调度器。

设备驱动是图 1.4 中所示的第 3 个组成部分。它们给硬件设备上的外设提供了一致的软件接口。

最后,一个应用程序完成一个设备所需的某个任务。例如,移动电话可能需要一个日志的应用程序。操作系统可以对同一个设备上运行的多个应用程序进行统一管理。

各个软件部分都可在 ROM 或 RAM 中运行。ROM 代码是固定在设备上的(例如初始化代码),被称之为固件(*firmware*)。

1.4.1 初始化(启动)代码

初始化代码(或启动代码)使处理器从复位状态进入到操作系统能够运行的状态。它通常须配置存储器控制器、处理器 cache 和初始化一些设备。在一个简单的系统中,操作系统可被一个简单的任务调度器或调试监控器所代替。

初始化代码在把控制权交给操作系统之前,须处理许多管理任务。我们可以把这些不同的任务划分为 3 个阶段:初始化硬件配置、诊断和引导。

初始化硬件配置包括设置目标平台,使之能够引导一个映像文件(image,后续执行的二进制代码)。尽管目标平台复位时自己有一个标准的配置,但是这个配置通常须修改,以满足被引导的映像文件的需求。例如,存储系统通常需要重新组织存储器映射(memory map),如例 1.1 所述。

诊断通常包含在初始化代码中。诊断代码用来检测系统,通过测试硬件目标来检测其

工作是否正常。同时它也检测标准的系统相关的事件。测试发生在软件产品完成后,因此这种测试对于生产制造是非常重要的。诊断代码的主要目的是识别和隔离故障。

引导过程包括了装载一个映像文件并将控制权交给它。如果系统必须引导不同的操作系统或者同一个操作系统的不同版本,那么引导过程本身就可能很复杂。

启动一个映像文件是最后一个阶段,但首先必须装载这个映像文件。装载一个映像文件的过程可以是拷贝包括代码和数据的整个程序到 RAM 中,也可以只拷贝包含易变(volatile)变量的数据区到 RAM 中。一旦启动,系统通过更改程序计数器(pc)指向映像文件的起始地址,将控制权交出。

有时,为了减小映像文件,映像文件会是压缩过的。在这种情况下,当映像文件被装载,或者当控制权递交给它时,映像文件要被解压。

【例 1.1】 初始化或组织存储器是初始化代码中的一个重要部分,因为许多操作系统在开始运行之前,希望了解存储器的组织情况。

图 1.5 对比了存储器在重新组织前/后的情况。对于基于 ARM 的嵌入式系统来说,通常都会提供存储器重映射,因为它允许系统上电后立刻从 ROM 中开始运行初始化代码。然后,初始化代码会重新定义或重构存储器映射,把 RAM 空间放在地址 0x00000000。这一步很重要,因为这样异常向量表就在 RAM 中,并可以被程序改写了。我们将在 2.4 节中详细讨论向量表。

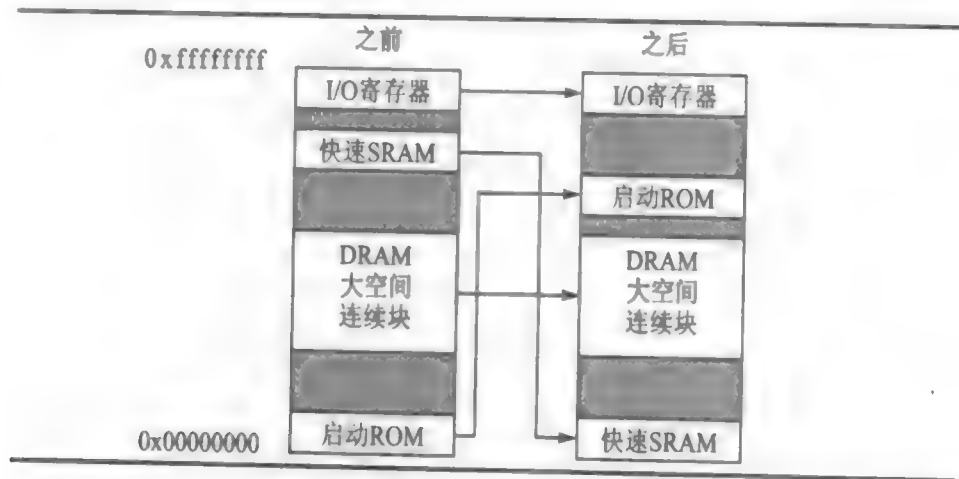


图 1.5 存储器重映射(remapping)

1.4.2 操作系统

初始化过程为操作系统进行控制准备好了硬件。操作系统组织系统资源:外设、存储器和处理时间。有了操作系统的管理,不同的应用程序在操作系统环境下就可以高效地使用这些资源。

ARM 嵌入式系统开发

ARM 处理器支持超过 50 种操作系统,可以把操作系统划分为两大类:实时操作系统(RTOS)和平台操作系统。

RTOS 保证对事件的响应时间。不同的操作系统对系统的响应时间有不同的控制。一个硬实时的应用需要一个完全被保证的响应时间;相反,一个软实时的应用只需要一个较好的响应时间,但是如果响应超时,则系统性能将会大大下降。运行 RTOS 的系统通常没有辅助存储器。

平台操作系统需要一个存储管理单元(MMU)来管理庞大的非实时应用,而且通常都有辅助存储器。Linux 操作系统就是平台操作系统的一个典型例子。

这两类操作系统也并非相互排外的,有一些操作系统使用 ARM 内核,带有内存管理单元,同时也具有实时特性。ARM 针对每一类操作系统都已经开发出一系列的处理器核。

1.4.3 应用程序

操作系统调度应用程序——为处理某个特定任务的代码。一个应用程序完成一个处理任务;操作系统控制整个运行环境。一个嵌入式系统可以只有一个活动的应用,也可以好几个应用同时在运行。

ARM 处理器被应用到众多的市场领域,包括网络、汽车、移动和消费类电子设备、海量存储设备及图像等。每一个领域中,又可以找到 ARM 的很多应用。

例如,在网络应用中,比如家庭网关、高速因特网通信中的 DSL 调制解调器以及 802.11 无线通信等,都存在 ARM 处理器。移动电话使移动式设备市场成为 ARM 处理器应用最广泛的领域。在海量存储设备中也有 ARM 处理器的应用,比如硬盘驱动器和图像产品,如喷墨打印机——这些应用都是成本敏感和数量众多的。

然而,在需要尖端高性能的应用中,并没有 ARM 处理器的身影。这是因为这些应用通常数量很少而成本很高,ARM 公司并不以这种类型的应用为目标进行设计。

1.5 总 结

纯粹的 RISC 是以高性能为主要目标的,但 ARM 采用的是一种改进的 RISC 设计思想,其目标是较高的代码密度和较低的功耗。一个嵌入式系统通常包含了一个处理器核,周围有 Cache、存储器和外设。操作系统控制整个系统,管理应用程序任务。

RISC 设计思想的关键是通过简化指令的复杂度来提高性能,使用流水线来加速指令的处理,提供大量的寄存器来存储数据,并使用 load-store 结构。

ARM 设计思想也包含了一些非 RISC 的观念或方法:

- 允许一些特定指令的执行周期数可变,以降低功耗,减小面积和代码尺寸;
- 增加了桶形移位器来扩展某些指令的功能;
- 使用 16 位的 *Thumb* 指令集来提高代码密度;
- 使用条件执行指令来提高代码密度和性能;
- 使用增强指令来实现数据信号处理的功能。

一个基于 ARM 的嵌入式系统通常包含以下的硬件组成部分:集成在芯片上的 *ARM* 处理器;程序员通过存储器映射地址的寄存器来访问的外设;一种特殊类型的外设,被称为控制器,使嵌入式系统可实现诸如存储器管理和中断控制等高层功能;片内的 *AMBA* 总线把处理器和外设连接在一起。

一个嵌入式系统包含以下的软件组成部分:初始化代码配置硬件到一个确定的状态;初始化成功后,操作系统就能够被装载和执行,操作系统提供一个通用的编程环境,以便各种应用能高效地使用系统的硬件资源;设备驱动给外设提供一个标准的程序接口;一个应用程序完成嵌入式系统的某个特定任务。

第 2 章

ARM 处理器基础

- 寄存器
- 当前程序状态寄存器
- 流水线
- 异常、中断及向量表
- 内核扩展
- 体系结构的不同版本
- ARM 处理器系列
- 总 结

第1章介绍了基于ARM处理器的嵌入式系统。本章将重点介绍处理器本身。首先,概述处理器内核以及数据在内核各组成部分之间是如何移动的,从软件开发者的角度描述ARM处理器的编程模型,这可以说明处理器内核的功能及其不同部分之间的相互作用;还将介绍基于ARM处理器的内核扩充,内核扩充不仅扩展了指令集,而且有效地组织和加速了主存(main memory);还将从ARM内核的命名规则简述ARM内核结构的不同版本(revision),并将按照时间顺序介绍ARM指令集结构的变化;最后将按照ARM处理器内核系列的分类来介绍其结构实现。

程序员可把ARM内核看作是由数据总线连接的各功能单元组成的集合,如图2.1所示。这里箭头代表了数据的流向,直线代表了总线,方框表示操作单元或存储区域。这个图不仅说明了数据流向,也说明了组成ARM内核的各个逻辑要素。

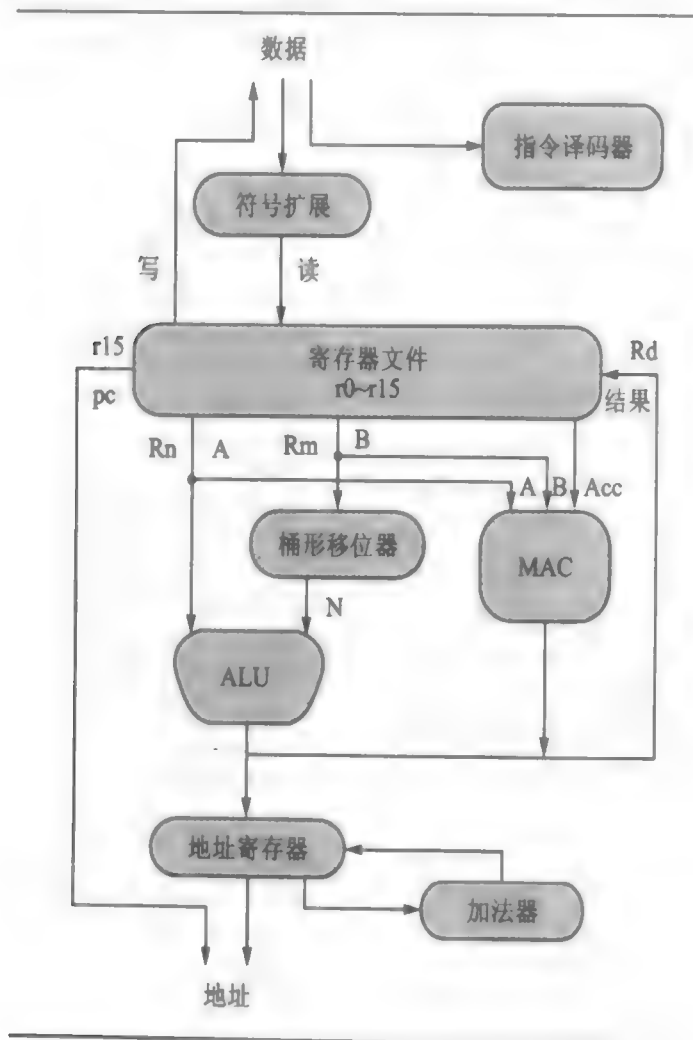


图 2.1 ARM 内核的数据流模型

数据通过数据总线进入处理器核,这里所指的数据可能是一条要执行的指令或一个数据项。图2.1显示了一个冯·诺伊曼(von Neumann)结构的ARM实现——数据和指令共

ARM 嵌入式系统开发

享同一总线;与此相反,哈佛(Harvard)结构的 ARM 实现使用二条不同的总线。

指令译码器在指令执行前先将它们翻译。每一条可执行指令都属于一个特定的指令集。

与所有的 RISC 处理器一样,ARM 处理器采用 load-store 体系结构。这就意味着它只有两种类型的指令用于把数据移入/移出处理器:load 指令从存储器复制数据到内核的寄存器;反过来,store 指令从寄存器里复制数据到存储器。没有直接操作存储器数据的数据处理指令,因此,数据处理只能在寄存器里进行。

数据项存储在寄存器文件里——一组 32 位的寄存器存储体(storage bank)。由于 ARM 内核是 32 位处理器,大部分指令认为寄存器中保存的是 32 位有符号或无符号数。当从存储器读取数据至一个寄存器时,符号扩展硬件会把 8 位和 16 位的有符号数转换成 32 位。

典型的 ARM 指令通常有 2 个源寄存器 Rn 和 Rm、1 个结果或目的寄存器 Rd。源操作数分别通过内部总线 A 和 B 从寄存器文件中读出。

ALU(算术逻辑单元)或 MAC(乘累加单元)通过总线 A 和 B 得到寄存器值 Rn 和 Rm,并计算出一个结果。数据处理指令直接把 Rd 中的计算结果写到寄存器文件。load-store 指令使用 ALU 来产生一个地址,这个地址将被保存到地址寄存器并发送到地址总线上。

ARM 的一个重要特征是,寄存器 Rm 可以选择在进入 ALU 前是否先经过桶形移位器预处理。桶形移位器和 ALU 协作可以计算较大范围的表达式和地址。

在经过有关功能单元后,Rd 寄存器里的结果值通过结果总线(result bus)写回到寄存器文件。对于 load-store 指令,在内核从下一个连续的存储器单元装载数据到下一个寄存器,或写下下一个寄存器的值到下一个连续的存储器单元之前,地址加法器会自动更新地址寄存器。处理器连续执行指令,直到发生异常或中断而改变了正常的执行流。

对处理器内核有了一个总体认识后,接下来我们将详细介绍处理器的各关键部件:寄存器、当前程序状态寄存器(cpsr)及流水线。

2.1 寄存器

通用寄存器可保存数据和地址。它们用字母 r 为前缀加该寄存器的序号来标识。例如:寄存器 4 可表示成 r4。图 2.2 列出了在用户模式(user mode,一种受限模式,通常用于执行应用程序)下的有效活动寄存器。处理器可以在 7 种不同的模式下运行,我们将会作简要介绍。所有的寄存器都是 32 位的。

最多可有 18 个活动寄存器:16 个数据寄存器和 2 个处理器状态寄存器。程序员可见的数据寄存器是 r0~r15。

ARM 处理器为特殊的任务或专门的功能指定了 3 个寄存器: r13, r14 和 r15。它们通常被赋予不同的标识,以区别于其它寄存器。

在图 2.2 里,有阴影的寄存器是有特殊用途的寄存器:

- 寄存器 r13 通常用作堆栈指针(sp),保存当前处理器模式的堆栈的栈顶;
- 寄存器 r14 又被称为链接寄存器(lr),保存调用子程序的返回地址;
- 寄存器 r15 是程序计数器(pc),其内容是处理器要取的下一条指令的地址。

根据具体的应用场合,寄存器 r13 和 r14 也可以用作通用寄存器,有时这会特别有用,因为在处理器模式改变时,这些寄存器是被分组备份的。但是若处理器上运行有任何形式的操作系统,把 r13 当作通用寄存器是很危险的,这是因为操作系统通常认为 r13 始终指向一个有效的栈结构。

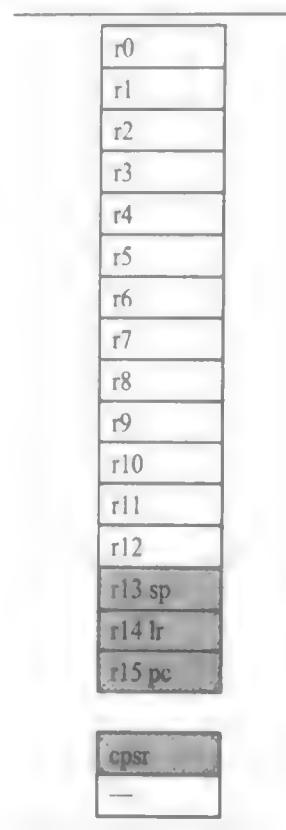


图 2.2 在用户模式下的有效寄存器

在 ARM 状态下,寄存器 r0~r13 是正交的(*orthogonal*)——任何指令如果可使用 r0,那么也就可使用其它寄存器。但是,有些指令是以特殊的方式来对待 r14 和 r15 的。

除了 16 个数据寄存器,还有 2 个程序状态寄存器: cpsr 和 spsr(分别是当前和备份的程序状态寄存器)。

寄存器文件包含所有程序员可利用的寄存器。哪些寄存器对程序员是可见的,取决于当前的处理器模式。

2.2 当前程序状态寄存器

ARM 内核使用 cpsr 来监视和控制内部的操作。cpsr 是寄存器文件中一个专用的 32 位寄存器。图 2.3 说明了一般程序状态寄存器的基本格式。

注意: 阴影部分是为将来扩展保留的。

cpsr 分为 4 个 8 位区域:标志域、状态域、扩展域和控制域。在目前的设计中,扩展和状态域保留,以便将来使用。控制域包含处理器模式、状态和中断屏蔽位;标志域包含条件标志位。

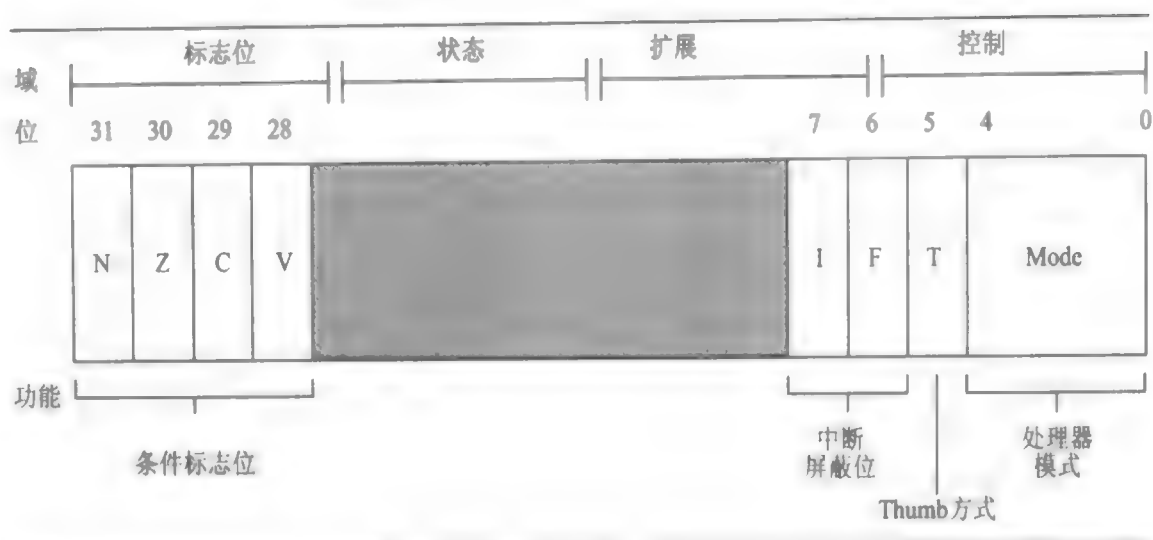


图 2.3 通用程序状态寄存器 (psr)

一些 ARM 处理器内核有额外的位分配。例如,在标志域里的 J 位只存在于 Jazelle 使能的处理器中,该类处理器可执行 8 位的指令。我们将在 2.2.3 小节更多地讨论 Jazelle。未来的设计中,极有可能为新功能的监测和控制分配额外的位。

关于 cpsr 的完整介绍请参照附录 B。

2.2.1 处理器模式

处理器模式决定了哪些寄存器是活动的以及对 cpsr 的访问权。处理器模式要么是特权模式,要么是非特权模式。特权模式允许对 cpsr 的完全读/写访问;与此相反,非特权模式只允许对 cpsr 的控制域进行读访问,但允许对条件标志的读/写访问。

具体共有 7 种处理器模式。6 种特权模式:中止(abort)模式、中断(interrupt request)模式、快速中断(fast interrupt request)模式、管理(supervisor)模式、系统(system)模式、未定义(undefined)模式;1 种非特权模式——用户(user)模式。

当处理器访问存储器失败时,进入数据访问中止模式(abort);中断模式和快速中断模式分别对 ARM 处理器两种不同级别的中断作出响应;处理器复位以后,进入管理模式,操作系统内核也通常处于这种模式;系统模式是一种特殊的用户模式,允许对 cpsr 的完全读/写访问;当处理器遇到没有定义的指令或处理器不支持该指令时,进入未定义模式;用户模式运行应用程序。

2.2.2 分组寄存器

图 2.4 列出了寄存器文件里所有 37 个寄存器。当然,在不同的时刻有 20 个寄存器对

程序是隐藏的。这些寄存器被称为**分组寄存器**，即图中所示的阴影部分。只有当处理器处于某种特定模式时，它们才有效，例如，中止模式使用分组寄存器 `r13_abt`、`r14_abt` 和 `spsr_abt`。一种特定模式所对应的分组寄存器采用这种方法来表示：寄存器_模式标记。模式标记可以是：`fiq`、`irq`、`svc`、`undef` 及 `abt` 等。

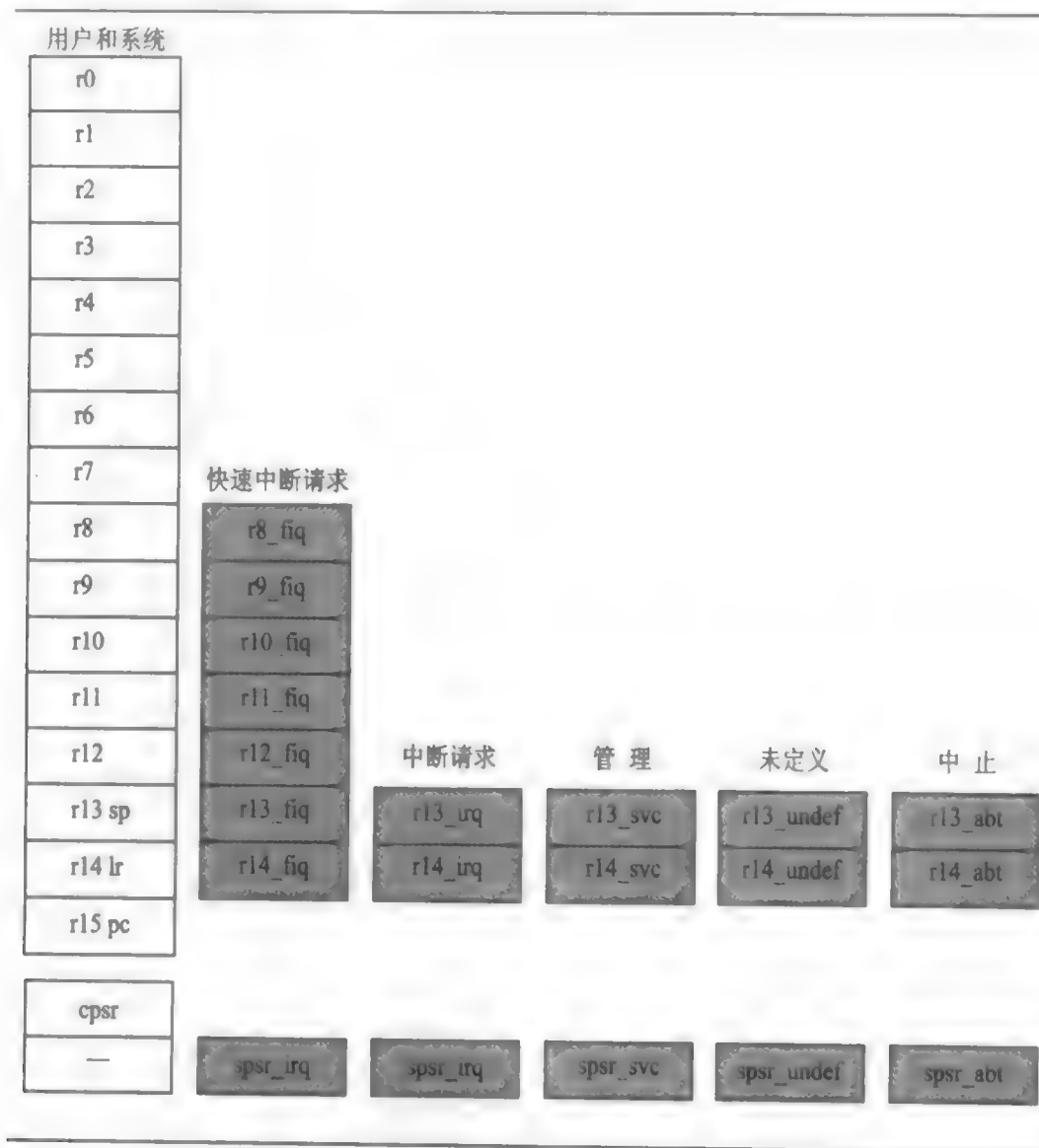


图 2.4 全部的 ARM 寄存器

除用户模式外，每一种处理器模式都可通过改写 `cpsr` 中的模式位来改变。除系统模式外，所有处理器模式都有一组各自的分组寄存器，它们是 16 个主要寄存器的子集。每个分组寄存器与一个用户模式的寄存器对应。如果改变处理器的模式，新模式的一个分组寄存器将取代原来模式的分组寄存器。

例如，当处理器处于中断模式时，执行的指令仍然可访问名字是 `r13` 和 `r14` 的寄存器，但实际上它们是分组寄存器 `r13_irq` 和 `r14_irq`。用户模式的 `r13_usr` 和 `r14_usr` 不会受到

ARM 嵌入式系统开发

影响。程序仍然可以正常访问寄存器 $r0 \sim r12$ 。

处理器模式既可以通过程序直接改写 $cpsr$ (处理器内核必须处于特权模式) 来切换, 也可以当内核对异常或中断响应时由硬件切换。下面的异常和中断会导致模式切换: 复位、外设中断请求、快速中断请求、软件中断、数据访问中止、预取指中止和未定义指令。异常和中断将挂起顺序指令的正常执行, 并跳转到一个特定的地址。

图 2.5 显示了当一个中断导致模式切换时所发生的情况。当一个外设向处理器核发出中断请求而产生中断时, 内核从用户模式切换到中断模式。这个改变使用户寄存器 $r13$ 和 $r14$ 被保护。用户寄存器 $r13$ 和 $r14$ 分别被寄存器 $r13_irq$ 和 $r14_irq$ 所代替。

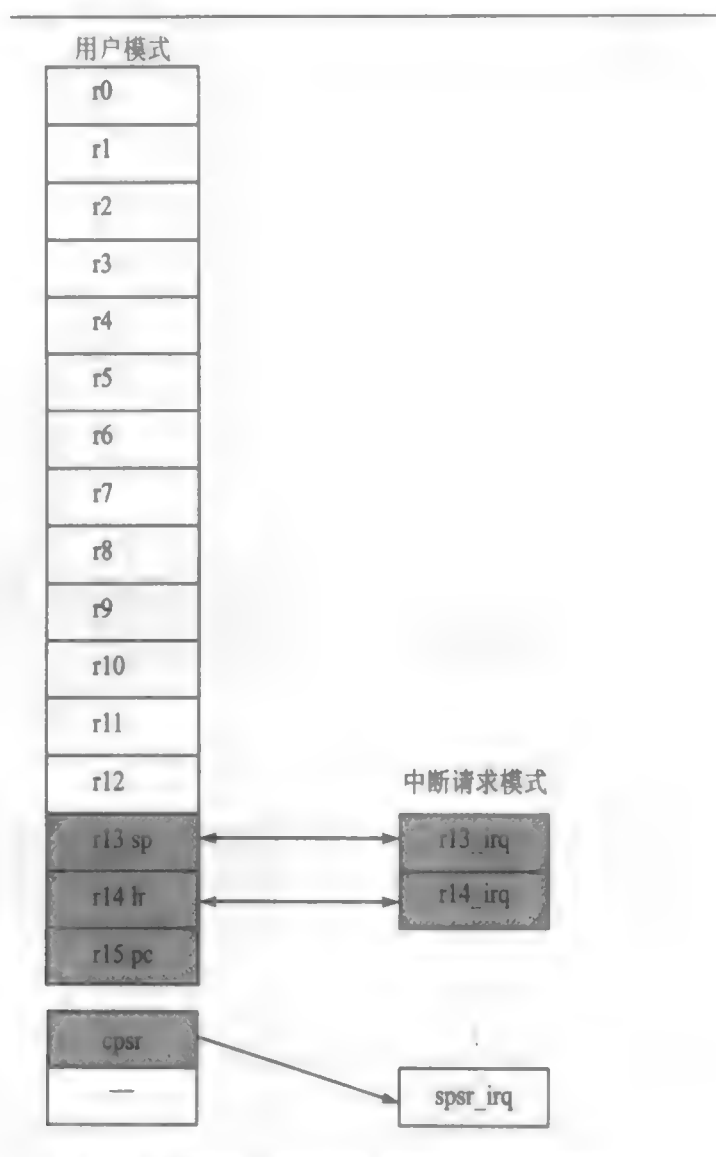


图 2.5 异常导致的模式改变

2 ARM 处理器基础

注意: r14_irq 保存了返回地址,而 r13_irq 保存的是中断模式的堆栈指针。

图 2.5 中也显示了在中断模式中的一个新寄存器:备份程序状态寄存器(spsr),它保存了先前处理器模式的 cpsr。从图中可以看到,cpsr 被复制到 spsr_irq。为了回到用户模式,须使用一条特殊的返回指令,指示内核从 spsr_irq 恢复原来的 cpsr,同时切换回先前受保护的用户寄存器 r13 和 r14。

注意:只能在特权模式下修改和读取 spsr,用户模式下没有 spsr。

另一个值得注意的特点是,当通过程序直接改写 cpsr 来切换模式时,cpsr 不会被复制到 spsr。只有当一个异常或中断发生时,才保存 cpsr。

图 2.3 显示了当前活动的处理器模式占据 cpsr 的低 5 位。上电后从管理模式开始。从一个特权模式开始是很有用的,因为初始化代码可以完全访问 cpsr,以设置其它各模式的堆栈。

表 2.1 列出了各种处理器模式,最后一列给出了 cpsr 中代表每一种处理器模式的二进制码。

表 2.1 处理器模式

模 式	缩 写	特 权	模式位[4:0]
中 止	abt	是	10111
快速中断请求	fiq	是	10001
中断请求	irq	是	10010
管 理	svc	是	10011
系 统	sys	是	11111
未定义	und	是	11011
用 户	usr	否	10000

2.2.3 状态和指令集

内核的状态决定了处理器将执行哪种指令集。有 3 种指令集:ARM,Thumb 和 Jazelle。只有当处理器处于 ARM 状态时,ARM 指令集才有效;同样,只有当处理器处于 Thumb 状态时,Thumb 指令才有效。一旦处于 Thumb 状态,处理器就纯粹执行 16 位的 Thumb 指令。不能把 ARM,Thumb 和 Jazelle 指令混合使用。

在 cpsr 中的 J(Jazelle)和 T(Thumb)位反映了处理器的状态。当 T 位和 J 位都为 0 时,处理器处于 ARM 状态,执行 ARM 指令,这是处理器上电时的默认状态。若 T 位置位,则处理器处于 Thumb 状态。为了改变状态,内核要执行专门的分支指令。表 2.2 对 ARM

ARM 嵌入式系统开发

和 Thumb 指令集的特征作了比较。

表 2.2 ARM 和 Thumb 指令集特征

项 目	ARM(cpsr T=0)	Thumb(cpsr T=1)
指令长度	32 位	16 位
内核指令	58 条	30 条
条件执行	大多数指令	只有分支指令
数据处理指令	访问桶形移位器和 ALU	独立的桶形移位器和 ALU 指令
程序状态寄存器	特权模式下可读/写	不能直接访问
寄存器使用	15 个通用寄存器+pc	8 个通用寄存器+7 个高寄存器+pc

注：条件执行可参见 2.2.6 小节。

ARM 设计者引进了第 3 种指令集,被称为 Jazelle。Jazelle 执行 8 位指令,它是一个软件与硬件的混合体,能够加速 Java 字节码(bytecodes)的执行。

为了执行 Java 字节码,需要 Jazelle 技术外加一个 Java 虚拟机的特殊修订版。特别要注意的是,Jazelle 的硬件部分只支持 Java 字节码的一个子集,其余的由软件仿真。

Jazelle 指令集是一个封闭的指令集,没有公开。表 2.3 给出了 Jazelle 指令集的一些特征。

表 2.3 Jazelle 指令集特征

项 目	Jazelle(cpsr T=0,J=1)
指令长度	8 位
内核指令	硬件完成超过 60%的 Java 字节代码,其余代码由软件完成

2.2.4 中断屏蔽

中断屏蔽位用来禁止某些中断请求来中断处理器。ARM 处理器内核有 2 个级别的中断请求——中断请求 IRQ 和快速中断请求 FIQ。

cpsr 有 2 个中断屏蔽位,位 7 和位 6(或 I 和 F)。它们分别控制 IRQ 和 FIQ 的中断屏蔽。I 位设置为 1 时,屏蔽 IRQ;同样,F 位设置为 1 时,屏蔽 FIQ。

2.2.5 条件标志

条件标志由比较指令或带有后缀 S 的 ALU 操作结果来设置。例如,如果一条 SUBS 减法指令产生了一个结果为 0 的寄存器值,那么 cpsr 中的 Z 标志就被置位。

在有 DSP 扩展的 ARM 内核中, Q 位表示增强的 DSP 指令是否发生了溢出或饱和。该位由硬件自动设置, 不能由硬件自动清除。要清除这一位, 必须由软件直接写 cpsr。

在有 Jazelle 扩展的处理器中, J 位反映了内核的状态: 若置位, 则内核处于 Jazelle 状态。J 位并不是都有用的, 它只对于某些处理器内核有效。为了使用 Jazelle, 需要从 ARM 公司和 Sun Microsystems 公司获得许可, 以得到额外的软件。

大多数 ARM 指令可根据条件标志位进行条件执行。表 2.4 列出了各条件标志位及如何产生置位的简短说明。这些标志位位于 cpsr 的高 5 位, 被用于条件执行。

表 2.4 条件标志

标志位	标志名	置位条件
Q	饱和	结果导致一个溢出和/或饱和
V	溢出	结果导致一个有符号数溢出
C	进位	结果导致一个无符号数进位
Z	零	结果是 0, 常用在指示相等与否
N	负数	结果的第 31 位为 1

图 2.6 给出了对于具有 DSP 扩展和 Jazelle 的 cpsr 典型值。本书中将采用符号来表示 cpsr 的值, 以便阅读。如果某一位置 1, 则用大写字母表示; 若某一位为 0, 则用小写字母表示。对于条件标志位, 大写字母表示该标志已被置位; 对于中断屏蔽位, 大写字母表示该中断被禁止。

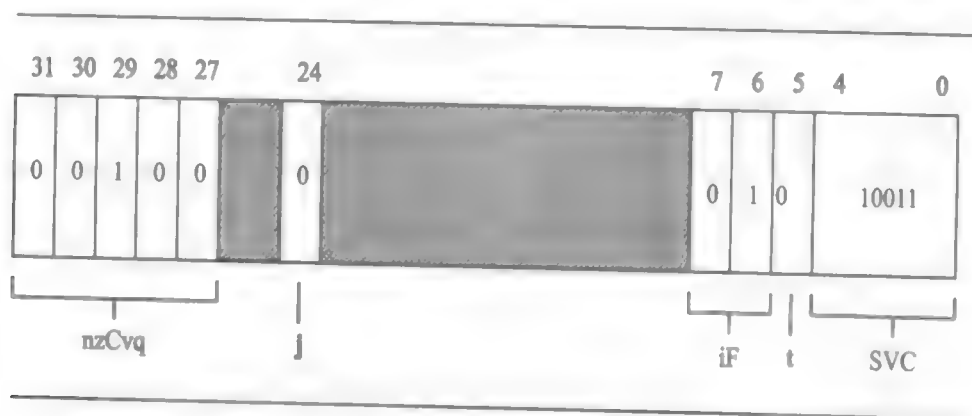


图 2.6 例子: cpsr = nzCvqjiFt_SVC

在图 2.6 所示的例子中, C 标志位是唯一置位的条件标志位, 其余标志位 `nzvq` 全都为 0。处理器处于 ARM 状态, 因为 J 和 T 位均为 0。IRQ 中断被使能(enabled), FIQ 中断被禁止。最后, 从图中可以看出, 处理器处于管理模式(SVC)下, 因为模式位[4:0]为 10011。

2.2.6 条件执行

条件执行控制内核是否将会执行一条指令。大多数指令都有一个条件属性,再根据条件标志位的情况,决定内核是否执行该指令。在执行前,处理器比较该条件属性和 cpsr 的条件标志位:如果匹配,指令就被执行;否则指令被忽略。

条件属性作为指令助记符的后缀被编码进指令。表 2.5 列出了条件执行代码助记符。若没有条件符号,则默认为无条件(AL)执行。

表 2.5 条件码助记符

助记符	名 称	条件标志位
EQ	相 等	Z
NE	不相等	z
CS HS	进位置位/无符号数大于或等于	C
CC LO	进位清除/无符号数小于	c
MI	负 数	N
PL	非负数	n
VS	溢 出	V
VC	无溢出	v
HI	无符号数大于	zc
LS	无符号数小于或等于	Z 或 c
GE	有符号数大于或等于	NV 或 nv
LT	有符号数小于	Nv 或 nV
GT	有符号数大于	NzV 或 nzv
LE	有符号数小于或等于	Z 或 Nv 或 nV
AL	无条件执行	忽 略

2.3 流水线

流水线是 RISC 处理器执行指令时采用的机制。使用流水线,可在取下一条指令的同时译码和执行其它指令,从而加快执行速度。可以把流水线看作是汽车生产线,每个阶段只完成一项专门的生产任务。

图 2.7 显示了一个 3 级流水线：

- 取指(fetch)——从存储器装载一条指令；
- 译码(decode)——识别将被执行的指令；
- 执行(execute)——处理指令并把结果写回寄存器。



图 2.7 ARM7 的 3 级流水线

图 2.8 通过一个简单的例子说明了流水线的机制。在第 1 个周期,内核从存储器取出指令 ADD;在第 2 个周期,内核取出指令 SUB,同时对 ADD 指令译码;在第 3 个周期,指令 SUB 和 ADD 都沿流水线移动,ADD 指令被执行,而 SUB 指令被译码,同时又取出 CMP 指令。流水线使得每个时钟周期就可以执行一条指令。

随着流水线深度(级数)的增加,每一段的工作量被削减了,这使得处理器可以工作在更高的频率,同时也改善了性能。但系统延时(latency)同样也增加了,这是因为在内核执行一条指令前,需要更多的周期来充填流水线。流水线级数的增加也意味着在某些段之间会产生数据相关。可使用指令调整技术来编写代码,以减少数据相关(关于指令调整的更多信息参见第 6 章)。

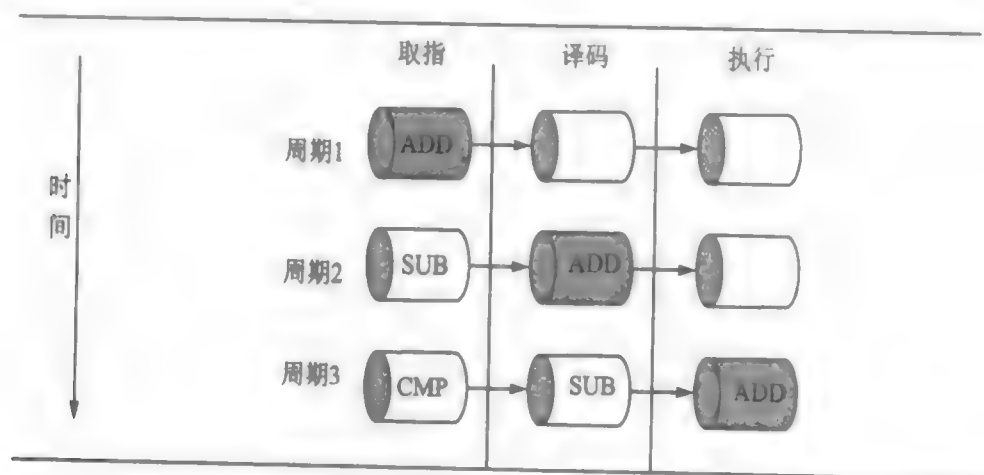


图 2.8 流水线指令顺序

每种 ARM 系列的流水线设计都有差异。例如,ARM9 内核把流水线深度增加到 5 级,如图 2.9 所示。ARM9 增加了存储器访问段和回写段,这使得 ARM9 的处理能力可达到平均 1.1 Dhrystone MIPS/MHz,与 ARM7 相比,指令吞吐量增加了约 13%。同时,ARM9 内核能达到的最高频率也更高。

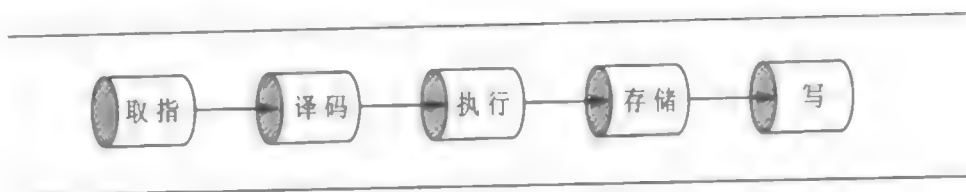


图 2.9 ARM9 的 5 级流水线

ARM10 更是把流水线的深度增加到 6 级,如图 2.10 所示。ARM10 的平均处理能力可达到 1.3 Dhrystone MIPS/MHz,与 ARM7 相比,指令吞吐量提高了约 34%;但同样有较大的系统延时。

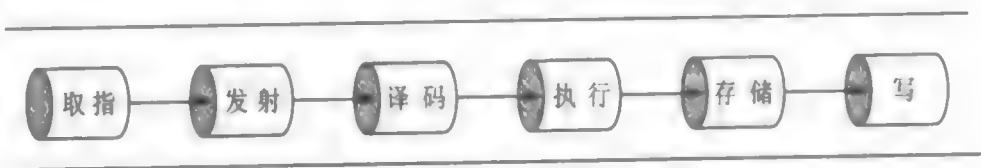


图 2.10 ARM10 的 6 级流水线

虽然 ARM9 和 ARM10 的流水线不同,但它们使用了与 ARM7 相同的流水线执行机制,因此 ARM7 上的代码也可以在 ARM9 和 ARM10 上运行。

流水线执行的特点

ARM 流水线的一条指令只有在完全通过“执行”阶段才被处理。例如,一条 ARM7 流水线(有 3 级)只有在取第 4 条指令时,第 1 条指令才完成执行。

图 2.11 反映了 ARM7 流水线上的指令顺序。MSR 指令用来使能 IRQ 中断,这只有在 MSR 指令完成流水线的“执行”阶段后才有效。它清除 cpsr 中的 I 位来使能 IRQ 中断。也就是说,一旦 ADD 指令进入“执行”阶段,IRQ 中断就被使能了。

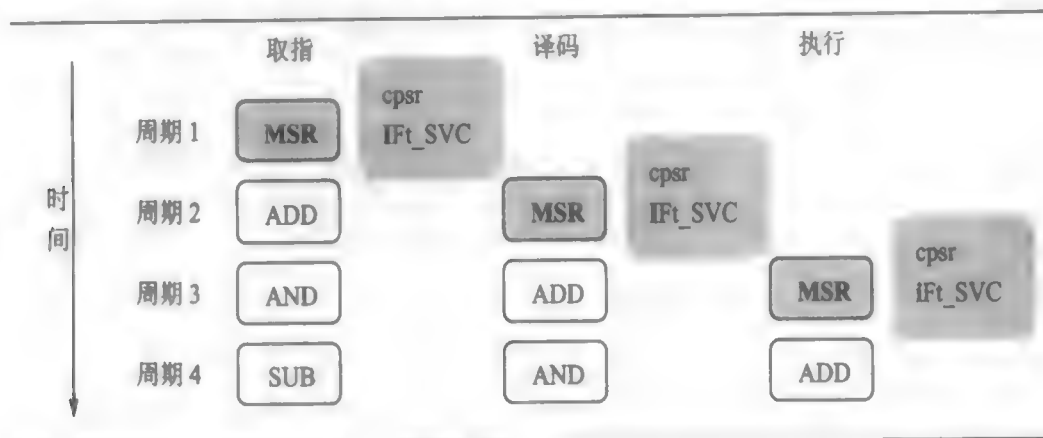


图 2.11 ARM 指令顺序

图 2.12 显示了流水线及程序计数器 pc 的使用情况。在指令“执行”阶段, pc 总是指向该指令地址加 8 字节的地址。换句话说, pc 总是指向当前正在执行的指令的地址再加 2 条指令的地址。当用 pc 来计算一个相对偏移量时, 这一点是很重要的, 并且它也是所有流水线的结构特征。

注意: 当处理器处于 Thumb 模式时, pc 的值为正在执行指令的地址加 4。

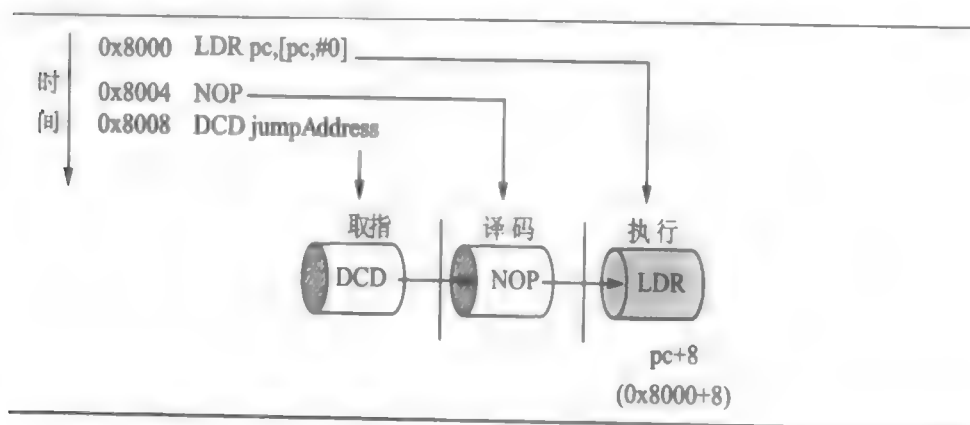


图 2.12 例子: $pc = address + 8$

另外, 还有 3 个值得注意的流水线特征:

第 1, 执行一条分支指令或直接修改 pc 而发生跳转时, 会使 ARM 内核清空流水线;

第 2, ARM10 使用分支预测技术, 通过预测可能的分支并在指令执行前装载新的分支地址, 从而减小了清空流水线的影响;

第 3, 即使产生了一个中断, 一条处于“执行”阶段的指令也将会完成。流水线里其它指令将会被放弃, 而处理器将从向量表的适当入口开始填充流水线。

27

2.4 异常、中断及向量表

当一个异常或中断发生时, 处理器会把 pc 设置为一个特定的存储器地址。这一地址放在一个被称为向量表(vector table)的特定的地址范围内。向量表的入口是一些跳转指令, 跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表(一组 32 位字)保留的。在有些处理器中, 向量表可以选择定位在存储空间的更高地址(从偏移量 0xffff0000 开始)。操作系统, 如 Linux 和 Microsoft 的嵌入式操作系统, 就可以利用这一特性。

当一个异常或中断发生时, 处理器挂起正常的执行转而从向量表(见表 2.6)装载指令。每一个向量表入口包含一条指向一个特定子程序的跳转指令。

表 2.6 向量表

异常/中断	缩写	地址	高位地址
复位	RESET	0x00000000	0xffff0000
未定义指令	UNDEF	0x00000004	0xffff0004
软件中断	SWI	0x00000008	0xffff0008
预取指中止	PABT	0x0000000c	0xffff000c
数据中止	DABT	0x00000010	0xffff0010
保留	—	0x00000014	0xffff0014
中断请求	IRQ	0x00000018	0xffff0018
快速中断请求	FIQ	0x0000001c	0xffff001c

- **复位向量**是处理器上电后执行的第一条指令的位置。这条指令使处理器跳转到初始化代码处。
- **未定义指令向量**是在处理器不能对一条指令译码时使用的。
- **软件中断向量**是执行 SWI 指令时被调用的。SWI 指令经常被用作调用一个操作系统例程的机制。
- **预取指中止向量**发生在处理器试图从一个未获得正确访问权限的地址去取指时,实际上中止发生在“译码”阶段。
- **数据中止向量**与预取指中止类似,发生在一条指令试图访问未获得正确访问权限的数据存储器时。
- **中断请求向量**是用于外部硬件(外设)中断处理器的正常执行流。只有当 cpsr 中的 IRQ 位未被屏蔽时才能发生。
- **快速中断请求向量**类似于中断请求,是为要求更短的中断响应时间的硬件保留的。只有当 cpsr 中的 FIQ 位未被屏蔽时才能发生。

2.5 内核扩展

本节包含的硬件扩展是置于 ARM 内核外围的标准组件。它们可以改善性能,管理资源以及提供额外的功能,为处理特殊的应用提供了灵活性。每个 ARM 系列都有不同的扩展。

有 3 种硬件扩展位于内核周围: cache 和紧耦合存储器 TCM(Tightly Coupled Memory)、存储管理及协处理器接口。

2.5.1 cache 和紧耦合存储器

cache 是位于主存储器和内核之间的快速存储器。它允许从某些存储器中更高效地取指。有了 cache, 处理器内核就能够在大多数时间全速运行而无须等待低速的外部存储器访问。大多数基于 ARM 的嵌入式系统使用处理器内部的一级 cache, 也有许多小的嵌入式系统不需要 cache 带来的性能上的改善。

ARM 有两种形式的 cache。第一种形式是针对冯·诺伊曼结构的内核。它把数据和指令放在一个统一的 cache 里, 如图 2.13 所示。为了简单起见, 我们把存储器与 AMBA 的连接部分统称为逻辑与控制。

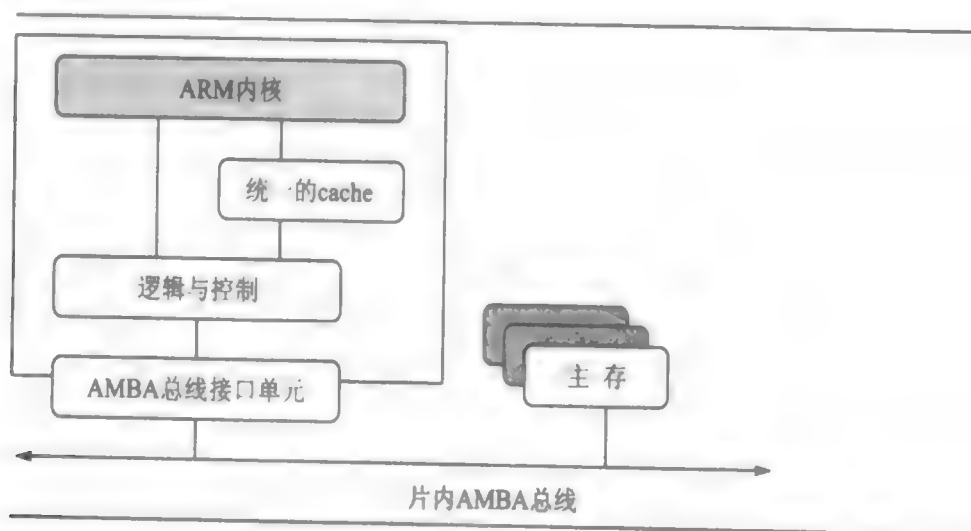


图 2.13 带 Cache 的简化冯·诺伊曼结构

第二种形式是针对哈佛结构的内核, 有独立的指令 cache 和数据 cache。

cache 改善了系统的整体性能, 但也使程序的执行时间变得不可预测。对于实时系统来说, 代码执行的确定性——装载和存储指令或数据的时间必须是可预测的, 这一点至关重要。使用称为紧耦合存储器 TCM 的存储器就可以实现。TCM 是一种快速 SRAM, 它紧挨内核, 并且保证取指或数据操作的时钟周期数——这对于一些要求确定行为的实时算法是很重要的。TCM 位于存储器地址映射中, 可作为快速存储器来访问。一个带 TCM 的处理器内核的例子如图 2.14 所示。

把上述两项技术结合, ARM 处理器既能够改善性能, 又能够获得可预测的实时响应。图 2.15 显示了一个结合了 cache 和 TCM 的内核例子。

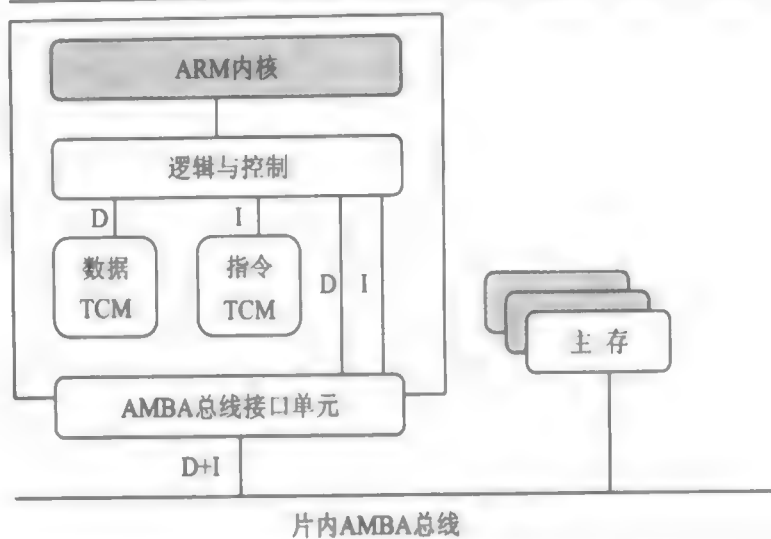


图 2.14 带 TCM 的简化哈佛结构

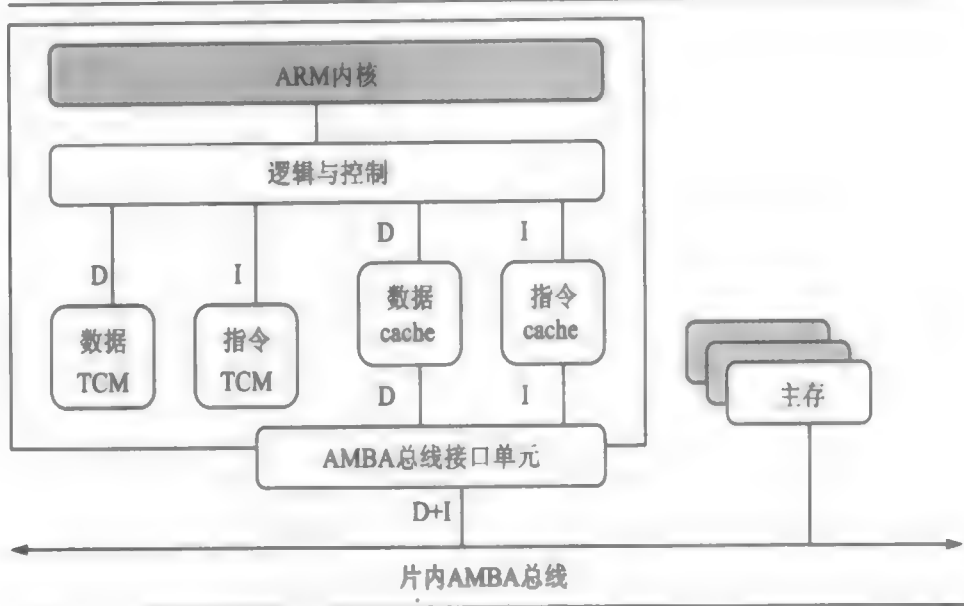


图 2.15 带 TCM 和 cache 的简化哈佛结构

2.5.2 存储管理

嵌入式系统通常使用多个存储设备,因此有必要实施某种策略来组织管理这些设备,并保护系统,避免一些应用非法访问硬件。可以使用存储器管理硬件来实现这些功能。

ARM 内核的存储器管理硬件有 3 种不同类型——没有提供扩展,没有硬件保护(无保护);提供有限保护的存储器保护单元(MPU);提供全面保护的存储器管理单元(MMU)。

- **无保护存储器**是固定的,只能提供非常有限的灵活性。它通常用于小的、简单的嵌入式系统,这种系统由于其应用特点而不要求存储器保护。
- **MPU** 使用一个只用到少量存储区域的简单系统。这些区域由一组特殊的协处理器寄存器控制,每一个区域定义了专门的访问权限。这种类型的存储器管理,适用于要求有存储器保护但没有复杂存储器映射的系统。关于 MPU 将在第 13 章中作进一步介绍。
- **MMU** 是 ARM 上最广泛的存储器管理硬件。MMU 使用一组转化表,以提供精细的存储器控制。这些表保存在主存里,并且提供虚拟地址与物理地址的映射和访问权限。MMU 适用于支持多任务的复杂操作系统平台。关于 MMU 将在第 14 章中作进一步介绍。

2.5.3 协处理器

协处理器可以附属于 ARM 处理器。一个协处理器通过扩展指令集或提供配置寄存器来扩展内核处理功能。一个或多个协处理器可以通过协处理器接口与 ARM 内核相连。

协处理器可以通过一组专门的、提供 load-store 类型接口的 ARM 指令来访问。例如协处理器 15(CP15),ARM 处理器使用协处理器 15 的寄存器来控制 cache、TCM 和存储器管理。

协处理器也能通过提供一组专门的新指令来扩展指令集。例如,有一组专门的指令可以添加到标准 ARM 指令集中,以处理向量浮点(VFP)运算。

这些新指令是在 ARM 流水线的译码阶段被处理的。如果在译码阶段发现是一条协处理器指令,则把它送给相应的协处理器。如果该协处理器不存在,或不认识这条指令,则 ARM 认为发生了未定义指令异常。这也使得编程者可以用软件来仿真协处理器的行为(使用未定义指令异常服务子程序)。

2.6 体系结构的不同版本

每个 ARM 处理器都有一个特定的指令集架构 ISA,而一个 ISA 版本又可以有多种处理器实现。

ISA 随着嵌入式市场的需求而发展。ARM 公司精心规划该发展过程,使得在较早的架构版本上编写的代码也可以在后继版本上执行。

在解释 ISA 的发展过程前,我们先来介绍 ARM 处理器的命名规则。命名规则确定了处理器具有的相关功能特性。

2.6.1 命名规则

ARM 使用如图 2.16 所示的命名规则来描述一个处理器。在“ARM”后的字母和数字表明了一个处理器的功能特性。随着更多特性的增加,将来字母和数字的组合可能会改变。

注意: 命名规则不包含体系结构的版本信息。

ARM {x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{S}

x——系列

y——存储管理/保护单元

z——Cache

T——Thumb 16位译码器

D——JTAG调试器

M——快速乘法器

I——嵌入式跟踪宏单元

E——增强指令(基于TDMI)

J——Jazelle

F——向量浮点单元

S——可综合版本

图 2.16 ARM 命名规则

关于 ARM 命名规则,还有一些附加的要点:

- ARM7TDMI 之后的所有 ARM 内核,即使“ARM”标志后没有包含那些字符,也都包括了 TDMI 的功能特性。
- **处理器系列**是共享相同硬件特性的一组处理器的具体实现。例如,ARM7TDMI, ARM740T 和 ARM720T 都共享相同的系列特性,都属于 ARM7 系列。
- **JTAG**是由 IEEE1149.1 标准测试访问端口(standard test access port)和边界扫描结构来描述的。它是 ARM 用来发送和接收处理器内核与测试仪器之间调试信息的一系列协议。
- **嵌入式 ICE 宏单元**(EmbeddedICE macrocell)是建立在处理器内部、用来设置断点和观察点的调试硬件。
- **可综合的**,意味着处理器内核是以源代码形式提供的。这种源代码形式又可以被编译成一种易于 EDA 工具使用的形式。

2.6.2 体系结构的发展

自从第一个 ARM 处理器于 1985 年问世以来,ARM 体系结构一直在发展。表 2.7 列出了从最初的体系结构版本 1 到现在的版本 6 的那些显著的增强。ISA 的最明显变化之一是,在 ARMv4T(ARM7TDMI 处理器)中引入了 Thumb 指令集。

表 2.7 版本历史

版 本	内核实现范例	ISA 增强
ARMv1	ARM1	第一个 ARM 处理器 26 位寻址
ARMv2	ARM2	32 位乘法器 32 位协处理器支持
ARMv2a	ARM3	片上 cache 原子交换指令 协处理器 15 用于 cache 管理
ARMv3	ARM6 和 ARM7DI	32 位寻址 独立的 cpsr 和 spsr 新增模式——未定义指令和中止模式 MMU 支持——虚拟存储
ARMv3M	ARM7M	有符号和无符号长乘法指令
ARMv4	StrongARM	有符号和无符号半字与字节的 load-store 指令 新增模式——系统模式 为体系结构上定义的操作而保留 SWI 空间 不再支持 26 位寻址模式
ARMv4T	ARM7TDMI 和 ARM9T	Thumb
ARMv5TE	ARM9E 和 ARM10E	ARMv4T 的超集 增加 ARM 与 Thumb 状态之间切换的额外指令 增强乘法指令 额外的 DSP 类型指令 快速乘累加
ARMv5TEJ	ARM7EJ 和 ARM926EJ	Java 加速
ARMv6	ARM11	改进的多处理器指令 边界不对齐和混合大小端数据的处理 新的多媒体指令

表 2.8 总结了程序状态寄存器的各个部分及其在特定指令架构上对应功能的有效性。

ARM 嵌入式系统开发

“全部”指的是 ARMv4 及其以上的体系结构。

表 2.8 cpsr 的描述

部 分	位	架 构	描 述
模式位	4:0	全 部	处理器模式
T	5	ARMv4T	Thumb 状态
I & F	7,6	全 部	中断屏蔽位
J	24	ARMv5TEJ	Jazelle 状态
Q	27	ARMv5TE	条件标志位
V	28	全 部	条件标志位
C	29	全 部	条件标志位
Z	30	全 部	条件标志位
N	31	全 部	条件标志位

2.7 ARM 处理器系列

ARM 公司设计了许多处理器,它们可以根据使用内核的不同划分到各个系列中。系列划分是基于 ARM7,ARM9,ARM10 和 ARM11 内核的。后缀数字 7,9,10 和 11 表示不同的内核设计。数字的升序说明性能和复杂度的提高。ARM8 开发出来以后很快就被取代了。

表 2.9 显示了 ARM7,ARM9,ARM10 及 ARM11 内核之间一个粗略的属性比较。其所列的一些数据可能会有较大变化,这依赖于生产过程的类型和工艺,它们对工作频率(MHz)和功耗(W)也会产生直接影响。

表 2.9 ARM 系列属性的比较

项 目	ARM7	ARM9	ARM10	ARM11
流水线深度	3 级	5 级	6 级	8 级
典型频率/MHz	80	150	260	335
mW/MHz ^a	0.06 mW/MHz	0.19 mW/MHz (+cache)	0.5 mW/MHz (+cache)	0.4 mW/MHz (+cache)
MIPS ^b /MHz	0.97	1.1	1.3	1.2
架 构	冯·诺伊曼	哈 佛	哈 佛	哈 佛
乘法器	8×32	8×32	16×32	16×32

注: a——在相同的 0.13 μm 工艺上的 W/MHz;

b——MIPS 是 Dhrystone VAX MIPS。

在每个系列中,存储器管理、cache 和 TCM 处理器扩展也有多种变化。ARM 继续在可用的产品系列和每个系列内部的不同变种两方面做进一步开发。

还有一些执行 ARM ISA 的处理器,如 StrongARM 和 XScale,这些处理器是由特定的半导体公司独家生产的,如上述处理器的厂商 Intel。

表 2.10 总结了各种处理器的不同功能特性。下面将从 ARM7 系列开始,依次就 ARM 系列作进一步的说明。

表 2.10 ARM 处理器不同功能特性

CPU 核	MMU/MPU	cache	Jazelle	Thumb	ISA	E*
ARM7TDMI	无	无	否	是	v4T	否
ARM7EJ-S	无	无	是	是	v5TEJ	是
ARM720T	MMU	统一的 8K cache	否	是	v4T	否
ARM920T	MMU	独立的 16K/16K D+I cache	否	是	v4T	否
ARM922T	MMU	独立的 8K/8K D+I cache	否	是	v4T	否
ARM926EJ-S	MMU	独立——cache 与 TCM 可配置	是	是	v5TEJ	是
ARM940T	MPU	独立的 4K/4K D+I cache	否	是	v4T	否
ARM946E-S	MPU	独立——cache 与 TCM 可配置	否	是	v5TE	是
ARM966E-S	无	独立——cache 与 TCM 可配置	否	是	v5TE	是
ARM1020E	MMU	独立的 32K/32K D+I cache	否	是	v5TE	是
ARM1022E	MMU	独立的 16K/16K D+I cache	否	是	v5TE	是
ARM1026EJ-S	MMU	独立——cache 与 TCM 可配置	是	是	v5TE	是
ARM1036J-S	MMU	独立——cache 与 TCM 可配置	是	是	v6	是
ARM1136JF-S	MMU	独立——cache 与 TCM 可配置	是	是	v6	是

注: a——E 扩展提供了增强的乘法指令和饱和运算指令。

2.7.1 ARM7 系列

ARM7 内核是冯·诺伊曼体系结构,数据和指令使用同一条总线。内核有一条 3 级流水线,执行 ARMv4 指令集。

ARM7TDMI 是 ARM 公司于 1995 年推出的新系列中的第一个处理器内核。是目前一个非常流行的内核,已被用在许多 32 位嵌入式处理器上。它提供了非常好的性能-功耗比。ARM7TDMI 处理器内核已经许可给许多世界顶级半导体公司,它是第一个包括 Thumb 指令集、快速乘法指令和嵌入式 ICE 调试技术的内核。

ARM7 系列中一个重要的变化是 ARM7TDMI-S。ARM7TDMI-S 与标准

ARM 嵌入式系统开发

ARM7TDMI 有相同的操作特性,但它是可综合的(见 2.6.1 小节)。

ARM720T 是 ARM7 系列中最具灵活性的成员,因为它包含了一个 MMU。MMU 的存在意味着 ARM720T 能够处理 Linux 和 Microsoft 嵌入式操作系统(如 WinCE)。这一处理器还包括了一个 8 KB 的统一 cache(指令/数据混合 cache)。向量表可通过设置一个协处理器 15(CP15)寄存器来重定位到更高的地址。

另一个成员是 ARM7EJ-S 处理器,也是可综合的。ARM7EJ-S 与其它 ARM7 处理器有很大不同,因为它有一条 5 级流水线,并且执行 ARMv5TEJ 指令。这个版本是 ARM7 中惟一一个提供 Java 加速和增强指令,而没有任何存储器保护的处理器。

2.7.2 ARM9 系列

ARM9 系列于 1997 年问世。由于采用了 5 级指令流水线,ARM9 处理器能够运行在比 ARM7 更高的时钟频率上,改善了处理器的整体性能;存储器系统根据哈佛体系结构重新设计,区分了数据 D 和指令 I 总线。

ARM9 系列的第一个处理器是 ARM920T,包含独立的 D+I cache 和一个 MMU。这个处理器能够被用在要求有虚拟存储器(虚存)支持的操作系统上。ARM922T 是 ARM920T 的变种,只有一半大小的 D+I cache。

ARM940T 包括一个更小的 D+I cache 和一个 MPU。它是针对不要求运行平台操作系统的应用而设计的。ARM920T 和 ARM940T 都执行 v4T 架构指令。

ARM9 系列的下一个处理器是基于 ARM9E-S 内核的。这个内核是 ARM9 内核带有 E 扩展的一个可综合版本。它有 2 个变种:ARM946E-S 和 ARM966E-S。两者都执行 v5TE 架构指令。它们也支持可选的嵌入式跟踪宏单元(ETM),允许开发者实时跟踪处理器上指令和数据的执行。当调试对时间敏感(time-critical)的程序段时,这种方法非常重要。

ARM946E-S 包括 TCM、cache 和一个 MPU。TCM 和 cache 的大小可配置。该处理器是针对要求有确定的实时响应的嵌入式控制应用而设计的。而 ARM966E 有可配置的 TCM,但没有 MPU 和 cache 扩展。

ARM9 产品线的最新内核是 ARM926EJ-S 可综合的处理器内核,发布于 2000 年。它是针对小型便携式 Java 设备,诸如 3G 手机和个人数字助理(PDA)应用而设计的。ARM926EJ-S 是第一个包含 Jazelle 技术(可加速 Java 字节码的执行)的 ARM 处理器内核。它还有一个 MMU、可配置的 TCM 以及具有零或非零等待存储器的 D+I cache。

2.7.3 ARM10 系列

ARM10 发布于 1999 年,主要是针对高性能的设计。它把 ARM9 的流水线扩展到 6 级,也支持可选的向量浮点单元 VFP,对 ARM10 的流水线加入了第 7 段。VFP 明显增强了浮点运算的性能,并与 IEEE754.1985 浮点标准兼容。

ARM1020E 是第一个使用 ARM10E 内核的处理器。像 ARM9E 一样,它包括了增强的 E 指令。它有独立的 32 KB D+I cache、可选向量浮点单元 VFP 以及 MMU。ARM1020E 还有一个双 64 位总线接口,以改善性能。

ARM1026EJ-S 非常类似于 ARM926EJ-S,但同时具有 MPU 和 MMU。这一处理器具有 ARM10 的性能和 ARM926EJ-S 的灵活性。

2.7.4 ARM11 系列

ARM1136J-S 发布于 2003 年,是针对高性能和高能效应用而设计的。ARM1136J-S 是第一个执行 ARMv6 架构指令的处理器。它集成了一条具有独立的 load-store 和算术流水线的 8 级流水线*。ARMv6 指令包含了针对媒体处理的单指令流多数据流(SIMD)扩展,采用特殊的设计,以改善视频处理性能。

ARM1136JF-S 就是为了进行快速浮点运算,而在 ARM1136J-S 增加了向量浮点单元。

2.7.5 专用处理器

StrongARM 最初是 ARM 公司与 Digital Semiconductor 公司合作开发的,现在由 Intel 公司单独许可。在要求低功耗、高性能和 PDA 上的应用很广泛。它是哈佛结构,具有独立的 D+I cache。StrongARM 是第一个包含 5 级流水线的高性能 ARM 处理器,但它不支持 Thumb 指令集**。

Intel 的 XScale 是 Strong ARM 的后续产品,在性能上有显著改善。在本书写作之际,据报道 XScale 可运行在高达 1 GHz 的频率上。XScale 执行 v5TE 架构指令,它是哈佛结构的,类似于 StrongARM,也包含一个 MMU。

SC100 致力于性能指标的另一方面。它是特别针对低功耗的安全应用而设计的。SC100 是第一个基于 ARM7TDMI 内核、带有 MPU 的安全内核。它不仅内核小,而且有较

* 可参见图 15.4。——译者注

** Intel 公司已宣布停产 StrongARM。——译者注

低的电压和电流需求,对于智能卡应用颇具吸引力。

2.8 总 结

在本章中,我们把注意力集中在实际 ARM 处理器的硬件基础上。ARM 处理器可抽象成 8 个部件——ALU、桶形移位器、MAC、寄存器文件、指令译码器、地址寄存器、增量加法器和符号扩展。

ARM 有 3 个指令集:ARM,Thumb 和 Jazelle。寄存器文件包含 37 个寄存器,但是在任意时刻只有 17 或 18 个寄存器可以被访问;其余的根据处理器模式被保护。当前的处理器模式保存在 cpsr 中,同时它还保存了处理器内核的当前状况、中断屏蔽位、条件标志和状态,该状态决定了哪个指令集正在被执行。

一个 ARM 处理器由一个内核及周围的组件并通过总线连接起来。内核扩展包括:

- *cache* 可改善系统的总体性能;
- *TCM* 可改进具有确定性的实时响应;
- *存储管理* 用于组织存储器和保护系统资源;
- *协处理器* 用于扩展指令集和功能,协处理器 15 控制 *cache*、*TCM* 和存储器管理。

一个 ARM 处理器是一个特定指令集架构 ISA 的具体实现。自从第一个 ARM 处理器问世以来,ISA 就一直在不断改进、完善。各种处理器根据相似的特性,可以被划分在各个系列(ARM7,ARM9,ARM10 和 ARM11)中。

第 3 章

ARM 指令集

- 数据处理指令
- 分支指令
- load-store 指令
- 软件中断指令
- 程序状态寄存器指令
- 常量的装载
- ARMv5E 扩展
- 条件执行
- 总 结

ARM 嵌入式系统开发

本章介绍的指令集是基础知识,因为这些内容在本书的其它部分都会用到。所以,在深入讨论优化和高效算法之前,首先需要学习指令集。本章将介绍最普通和常用的 ARM 指令,这些内容是建立在前一章所涉及的 ARM 处理器基础知识基础上的。第 4 章将介绍 Thumb 指令集,附录 A 给出了所有的 ARM 指令的完整描述。

不同的 ARM 体系结构版本支持的指令是不同的,但是新的版本一般是增加指令并且保持指令的向后兼容。也就是说,在 ARMv4T 上写的代码在 ARMv5TE 处理器上也是可以运行的。表 3.1 列出了所有 ARMv5E 指令集架构 ISA 下支持的指令。这个 ISA 包括了所有 ARM 指令,以及在 ARM 指令集中一些新的功能特征。“ARM ISA”一列,列出了该指令是在哪个修定版本引入的。一些指令在后续的架构中已扩展了功能,例如 CDP 指令在 ARMv5 中有一个变体叫 CDP2。同样,有些指令,像 LDR,也有 ARMv5 的扩展,但不需要新的或扩展的助记符。

表 3.1 ARM 指令集

助记符	ARM ISA	说 明
ADC	v1	带进位的 32 位数加法
ADD	v1	32 位数相加
AND	v1	32 位数的逻辑与
B	v1	在 32M 空间内的相对跳转指令
BIC	v1	32 位数的逻辑位清零
BKPT	v5	断点指令
BL	v1	带链接的相对跳转指令
BLX	v5	带链接的切换跳转
BX	v4T	切换跳转
CDP CDP2	v2 v5	协处理器数据处理操作
CLZ	v5	零计数
CMN	v1	比较两个数的相反数
CMP	v1	32 位数比较
EOR	v1	32 位逻辑异或
LDC LDC2	v2 v5	从协处理器取一个或多个 32 位值
LDM	v1	从内存送多个 32 位字到 ARM 寄存器
LDR	v1 v4 v5E	从虚拟地址取一个单个的 32 位值
MCR MCR2 MCRR	v2 v5 v5E	从寄存器送数据到协处理器
MLA	v2	32 位乘累加
MOV	v1	传送一个 32 位数到寄存器

续表 3.1

助记符	ARM ISA	说 明
MRC MRC2 MRRC	v2 v5 v5E	从协处理器传送数据到寄存器
MRS	v3	把状态寄存器的值送到通用寄存器
MSR	v3	把通用寄存器的值传送到状态寄存器
MUL	v2	32 位乘
MVN	v1	把一个 32 位数的逻辑“非”送到寄存器
ORR	v1	32 位逻辑或
PLD	v5E	预装载提示指令
QADD	V5E	有符号 32 位饱和加
QDADD	V5E	有符号双 32 位饱和加
QSUB	v5E	有符号 32 位饱和减
QDSUB	v5E	有符号双 32 位饱和减
RSB	v1	逆向 32 位减法
RSC	v1	带进位的逆向 32 位减法
SBC	v1	带进位的 32 位减法
SMLAxy	v5E	有符号乘累加($(16 \text{ 位} \times 16 \text{ 位}) + 32 \text{ 位} = 32 \text{ 位}$)
SMLAL	v3M	64 位有符号乘累加($(32 \text{ 位} \times 32 \text{ 位}) + 64 \text{ 位} = 64 \text{ 位}$)
SMLALxy	v5E	64 位有符号乘累加($(16 \text{ 位} \times 16 \text{ 位}) + 64 \text{ 位} = 64 \text{ 位}$)
SMLAWy	v5E	有符号乘累加($((32 \text{ 位} \times 16 \text{ 位}) \gg 16 \text{ 位}) + 32 \text{ 位} = 32 \text{ 位}$)
SMULL	v3M	64 位有符号乘累加($32 \text{ 位} \times 32 \text{ 位} = 64 \text{ 位}$)
SMULxy	v5E	有符号乘($16 \text{ 位} \times 16 \text{ 位} = 32 \text{ 位}$)
SMULWy	v5E	有符号乘($32 \text{ 位} \times 16 \text{ 位} \gg 16 = 32 \text{ 位}$)
STC STC2	v2 v5	从协处理器中把一个或多个 32 位值存到内存
STM	v1	把多个 32 位的寄存器值存放到内存
STR	v1 v4 v5E	把寄存器的值存到一个内存的虚地址空间
SUB	v1	32 位减法
SWI	v1	软中断
SWP	v2a	把一个字或者一个字节和一个寄存器值交换
TEQ	v1	等值测试
TST	v1	位测试
UMLAL	v3M	64 位无符号乘累加($(32 \text{ 位} \times 32 \text{ 位}) + 64 \text{ 位} = 64 \text{ 位}$)
UMULL	v3M	64 位无符号乘法($32 \text{ 位} \times 32 \text{ 位} = 64 \text{ 位}$)



后面举例介绍指令操作时,使用 PRE 和 POST 条件,分别表示指令执行前和执行后的内存、寄存器情况。对于十六进制数字,使用 0x 前缀表示;二进制使用 0b 前缀。示例格式如下:

PRE 〈执行前条件〉

 〈指令〉

POST 〈执行后情况〉

在执行前、后的条件说明中,如果需要对内存进行说明,则使用的格式为:

`mem<data_size>[address]`

其含义是从 address 开始的 data_size 存储器位,例如 mem32[1024]表示从地址 1 KB 开始的 32 位数据值。

ARM 指令只对存放在寄存器的数据进行处理,对于存储器数据,只能使用 load 和 store 指令进行存取。ARM 指令通常带有 2 个或 3 个操作数,例如下面的加法指令 ADD,把存放在寄存器 r1 和 r2 中的值相加,然后把结果存到寄存器 r3。

指令语法	目标寄存器(Rd)	源寄存器 1(Rn)	源寄存器 2(Rm)
ADD r3,r1,r2	r3	r1	r2

ARM 指令可以划分为以下几类:数据处理指令、分支指令、load-store 指令、软件中断指令和程序状态寄存器指令。后续章节将分类介绍 ARM 指令的语法和功能。

3.1 数据处理指令

数据处理指令对于存放在寄存器中的数据进行操作,包括 MOVE(传送)指令、算术指令、逻辑指令、比较指令和乘法指令。大多数数据处理指令可以使用桶形移位器对其中的一个操作数进行预处理。

如果在数据处理指令前使用 S 前缀,指令的执行将会更新 cpsr 中的标志。MOVE 指令和逻辑指令会对进位标志 C、负数标志 N 及零标志 Z 产生影响。在最后一位移出时,桶形移位的结果将更新进位标志 C;N 标志根据操作结果的第 31 位进行设置;如果结果为零,那么 Z 标志就会被设置。

3.1.1 MOVE 指令

MOVE 指令是最简单的 ARM 指令,执行的结果就是把一个数 N 送到目标寄存器 Rd。

其中 N 可以是寄存器,也可以是立即数。MOVE 指令多用于设置初始值或者在寄存器间传送数据。

MOVE 指令语法:

<指令>{<cond>}{S} Rd, N

MOV	把一个 32 位数送到一个寄存器	Rd = N
MVN	把一个 32 位数的“非”送到一个寄存器	Rd = ~N

通常 N 是一个寄存器 Rm 或者是一个使用 # 前缀的常量,3.1.2 小节中的表 3.3 将会对第 2 个操作数 N 的取值做完整的描述。

【例 3.1】 MOVE 指令。

例子中 MOV 指令把寄存器 r5 的内容复制到 r7 中去。执行后寄存器 r7 中的 8 被 5 覆盖。

PRE

r5 = 5

r7 = 8

MOV r7, r5, r7 = r5

POST

r5 = 5

r7 = 5

3.1.2 桶形移位器

在例子 3.1 中, N 是一个寄存器。其实 N 不仅可以表示寄存器或者立即数,也可以是一个在数据处理指令使用前被桶形移位器预处理过的寄存器 Rm。

数据处理指令是在算术逻辑单元 ALU 中完成的。ARM 处理器一个显著的特征就是,可以在操作数进入 ALU 之前,对操作数进行指定位数的左移或者右移。这种功能明显增强了许多数据处理操作的灵活性。

有些数据处理指令并没有用到桶形移位器,例如 MUL, CLZ 和 QADD 指令等。

预处理或移位发生在该指令周期内。这对于把一个常量送入寄存器,被 2 的幂乘或除等操作是特别有用的。

为了对桶形移位器进行说明,以图 3.1 为例,在 MOVE 指令中增加移位操作。寄存器 Rn 在进入 ALU 前没有进行移位预处理操作, Rm 使用桶形移位器移位后(产生结果 N)进

入 ALU。

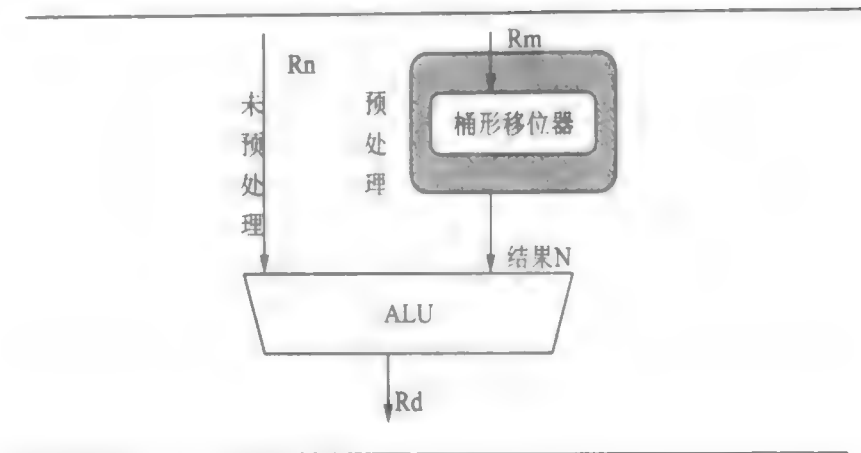


图 3.1 桶形移位器和 ALU

【例 3.2】在寄存器 Rm 送入目标寄存器之前,首先对它执行逻辑左移(LSL)。这就像在标准 C 中对寄存器使用移位操作“<<”。MOV 指令把桶形移位器操作的结果 N 放入寄存器 Rd,N 是 LSL 操作的结果。

PRE

r5 = 5

r7 = 8

MOV r7, r5, LSL #2 ;let r7 = r5 * 4 = (r5 << 2)

POST

r5 = 5

r7 = 20

这个例子中,寄存器 r5 乘以 4 后把结果放到寄存器 r7。

使用桶形移位器可执行 5 种不同的移位操作,如表 3.2 所列。

表 3.2 桶形移位器操作

助记符	说明	移位操作	结果	Y 值
LSL	逻辑左移	x LSL y	$x \ll y$	#0-31 or Rs
LSR	逻辑右移	x LSR y	$(\text{unsigned})x \gg y$	#1-32 or Rs
ASR	算术右移	x ASR y	$(\text{signed})x \gg y$	#1-32 or Rs
ROR	循环右移	x ROR y	$((\text{unsigned})x \gg y) (x \ll (32-y))$	#1-32 or Rs
RRX	扩展的循环右移	x RRX	$(c \text{ flag} \ll 31) ((\text{unsigned})x \gg 1)$	none

注: x 表示要进行移位操作的寄存器,y 表示要移位的位数。

图 3.2 是一个逻辑左移一位的操作,如位 0 移入位 1,位 0 清除,C 标志位被替换为移出寄存器的那一位。当 y 表示移位位数时,原来数的位 $(32-y)$ 将被移入 C 标志位,即对 C 标志进行替换。如果移位位数 $y>1$,那么执行移位量为 y 的移位和执行 y 次移位量为 1 的移位效果相同。

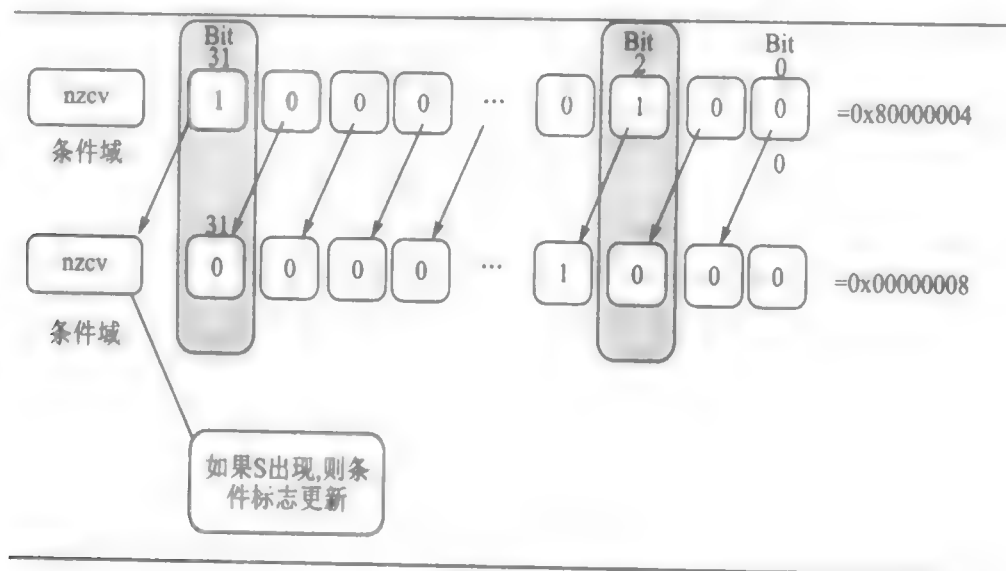


图 3.2 逻辑左移一位

【例 3.3】 一条 MOVS 指令,把寄存器 r1 左移一位送到寄存器 r0。
这相当于把 r1 乘以 2^1 。因为指令助记符中有 S 出现,故 cpsr 的 C 标志位会被更新。

PRE

```
cpsr = nzcvcIfT_USER
r0 = 0x00000000
r1 = 0x80000004
```

```
MOVS r0,r1,LSL #1
```

POST

```
cpsr = nvCvcIfT_USER
r0 = 0x00000008
r1 = 0x80000004
```

表 3.3 列出了在数据操作处理指令中可以使用的移位操作的语法。第 2 个操作数 N 可以是立即数和寄存器 R_m ,也可以是使用桶形移位器预处理过的寄存器。

表 3.3 数据处理指令的桶形移位操作语法

N	语 法
立即数	# immediate
寄存器	Rm
立即数逻辑左移	Rm, LSL # shift_imm
寄存器逻辑左移	Rm, LSL Rs
立即数逻辑右移	Rm, LSR # shift_imm
寄存器逻辑右移	Rm, LSR Rs
立即数算术逻辑右移	Rm, ASR # shift_imm
寄存器算术逻辑右移	Rm, ASR Rs
立即数循环右移	Rm, ROR # shift_imm
寄存器循环右移	Rm, ROR Rs
扩展循环右移	Rm, RRX

3.1.3 算术指令

算术指令用于实现 32 位有符号数或者无符号数的加法和减法操作。
语法为：

⟨指令⟩{⟨cond⟩}{S} Rd, Rn, N

ADC	32 位带进位加法	$Rd = Rn + N + \text{carry}$
ADD	32 位加法	$Rd = Rn + N$
RSB	32 位逆向减法	$Rd = N - Rn$
RSC	带进位的 32 位逆向减法	$Rd = N - Rn - !(\text{carry flag})$
SBC	带进位的 32 位减法	$Rd = Rn - N - !(\text{carry flag})$
SUB	32 位减法	$Rd = Rn - N$

注：N 是桶形移位器操作的结果，移位操作的语法见表 3.3。

【例 3.4】 使用减法指令把存储在 r1 中的值减去 r2 中的值，然后把结果存放在 r0 中。

PRE

r0 = 0x00000000

r1 = 0x00000002

r2 = 0x00000001

```
SUB r0,r1,r2
```

POST

```
r0 = 0x00000001
```

【例 3.5】 使用逆向减法指令 RSB 从常数 0 减去 r1,然后把结果存放在 r0 中。可使用这种方法对一个数进行取反操作。

PRE

```
r0 = 0x00000000
```

```
r1 = 0x00000077
```

```
RSB r0,r1,#0 ;Rd = 0x0 - r1
```

POST

```
r0 = -r1 = 0xffffffff89
```

【例 3.6】 SUBS 指令可以很方便地实现循环计数器的递减操作。

本例使用 SUBS 指令从寄存器 r1 中减去常量 1,然后把结果写回到 r1。

注意: cpsr 的 Z 和 C 位将会受到影响。

PRE

```
cpsr = nzcvcqifT_USER
```

```
r1 = 0x00000001
```

```
SUBS r1,r1,#1
```

POST

```
cpsr = nZCvcqifT_USER
```

```
r1 = 0x00000000
```

3.1.4 算术指令使用桶形移位器

在算术指令和逻辑指令中广泛使用的第 2 操作数的移位功能,是 ARM 指令集的一个非常显著的特征。例 3.7 是一个算术指令使用内嵌桶形移位器的示例,指令把存储在 r1 中的值乘以 3。

【例 3.7】 寄存器 r1 首先左移一位,其结果等于 r1 值的 2 倍;然后加法操作把 r1 和桶形移位器的结果相加;最终的结果放在寄存器 r0 中,r0 等于 $3 \times r1$ 。

PRE

```
r0 = 0x00000000
```

```
r1 = 0x00000005
```

```
ADD r0, r1, r1, LSL #1
```

POST

```
r0 = 0x0000000f
```

```
r1 = 0x00000005
```

3.1.5 逻辑指令

逻辑指令可对 2 个源操作数的对应位进行逻辑操作。

语法为：

⟨指令⟩{⟨cond⟩}{S} Rd, Rn, N

AND	32 位逻辑“与”	$Rd = Rn \& N$
ORR	32 位逻辑“或”	$Rd = Rn N$
EOR	32 位逻辑“异或”	$Rd = Rn \oplus N$
BIC	逻辑位清除 (AND NOT)	$Rd = Rn \& \sim N$

【例 3.8】 逻辑“或”操作。

把寄存器 r1 和 r2 进行“或”操作，然后把结果放到 r0。

PRE

```
r0 = 0x00000000
```

```
r1 = 0x02040608
```

```
r2 = 0x10305070
```

```
ORR r0, r1, r2
```

POST

```
r0 = 0x12345678
```

【例 3.9】 本例使用一个更复杂的逻辑指令——BIC 指令，它可以把一个逻辑位清零。

PRE

```
r1 = 0b1111
```

```

r2 = 0b0101
BIC r0, r1, r2

```

POST

```

r0 = 0b1010

```

上面代码的执行等价于： $Rd = Rn \text{ AND } \text{NOT}(N)$

在这个例子中，寄存器 $r2$ 中置 1 的位将清除 $r1$ 中对应位置的位。BIC 指令在清除状态位时是非常有用的，也经常用于改变 cpsr 中的中断屏蔽位。

只有当逻辑指令有 S 后缀时，指令才会更新 cpsr。这些指令可以像算术指令一样使用桶形移位第 2 操作数。

3.1.6 比较指令

比较指令通常用于把一个寄存器与一个 32 位的值进行比较或测试。比较指令根据结果更新 cpsr 的标志位，但不影响其它的寄存器。在设置标志位后，其它指令可通过条件执行来改变程序的执行流程。关于条件执行的更多信息，请参看 3.8 节。对于比较指令，不需要使用 S 后缀就可以改变标志位。

指令语法：

$\langle \text{指令} \rangle \{ \langle \text{cond} \rangle \} Rn, N$

CMN	取负比较	标记根据 $Rn+N$ 的值设置
CMP	比 较	标记根据 $Rn-N$ 的值设置
TEQ	等值测试	标记根据 $Rn \& N$ 的值设置
TST	位测试	标记根据 $Rn \& N$ 的值设置

注：N 是桶形移位器的操作结果，移位器操作的语法见表 3.3。

【例 3.10】 CMP 指令。

指令执行前， $r0$ 与 $r9$ 相等，z 标志是 0，用小写字母表示。执行后，z 标志变成 1，用大写字母表示，指示出比较结果相等。

PRE

```

cpsr = nzcvcqifl_USER
r0 = 4
r9 = 4

```

```
CMP r0, r9
```

POST

```
cpsr = nZcvqiFt_USER
```

比较指令 CMP 本质上就是一个不返回运算结果的减法指令；同样，TST 指令是一个没有保存结果的逻辑“与”操作；TEQ 则是一个逻辑“异或”操作。对于每个操作，不需要保存运算结果，只根据结果影响 cpsr。

必须记住：比较指令只改变 cpsr 中的条件标志，不影响参与比较的寄存器内容。

3.1.7 乘法指令

乘法指令把一对寄存器的内容相乘，然后根据指令类型把结果累加到其它的寄存器。长整型的“乘累加”要使用代表 64 位的一对寄存器，最终的结果放在一个目标寄存器或者一对寄存器中。

乘法指令的语法：

```
MLA {<cond>} {S} Rn, Rm, Rs, Rn
```

```
MUL {<cond>} {S} Rd, Rm, Rs
```

MLA	乘累加	$Rd = (Rm * Rs) + Rn$
MUL	乘法	$Rd = Rm * Rs$

```
<指令> {<cond>} {S} RdLo, RdHi, Rm, Rs
```

SMLAL	长整型有符号乘累加	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	长整型有符号乘法	$[RdHi, RdLo] = Rm * Rs$
UMLAL	长整型无符号乘累加	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	长整型无符号乘法	$[RdHi, RdLo] = Rm * Rs$

执行一个乘法指令需要的周期数取决于处理器的具体实现。对于某些处理器实现，需要的周期数还依赖于 Rs 的值。关于周期定时的更多细节，可参考附录 D。

【例 3.11】 一个简单的乘法指令：把寄存器 r1 和寄存器 r2 相乘，然后把结果放在寄存器 r0。本例中 r1=2, r2=2, 运算结果为 4, 放在寄存器 r0。

PRE

```
r0 = 0x00000000
```



```
r1 = 0x00000002
```

```
r2 = 0x00000002
```

```
MUL r0,r1,r2          ;r0 = r1 * r2
```

POST

```
r0 = 0x00000004
```

```
r1 = 0x00000002
```

```
r2 = 0x00000002
```

长整型乘法指令(SMLAL, SMULL, UMLAL 和 UMULL)产生 64 位的结果。由于结果太大,不能存放在一个 32 位的寄存器,所以把结果存放在 2 个 32 位的寄存器 RdLo 和 RdHi 中。RdLo 存放低 32 位,RdHi 存放高 32 位。例 3.12 是一个长整型乘法的例子。

【例 3.12】 把寄存器 r2 和 r3 的结果相乘,然后把结果放在 r0 和 r1 中。r0 存放低 32 位,r1 存放高 32 位。

PRE

```
r0 = 0x00000000
```

```
r1 = 0x00000000
```

```
r2 = 0xf0000002
```

```
r3 = 0x00000002
```

```
UMULL r0,r1,r2,r3      ;[r1,r0] = r2 * r3
```

POST

```
r0 = 0xe0000004      ;RdLo
```

```
r1 = 0x00000001      ;RdHi
```

3.2 分支指令

分支指令可改变程序的执行流程或者调用子程序。这种指令使得一个程序可以使用子程序、if - then - else 结构以及循环。执行流程的改变迫使程序计数器 pc 指向一个新的地址,ARMv5E 架构指令集包括以下 4 种不同的分支指令。

语法:

`B{<cond>} label`
`BL{<cond>} label`
`BX{<cond>} Rm`
`BLX{<cond>} label | Rm`

B	跳 转	pc=label
BL	带返回的跳转	pc=label lr=BL 后面的第一条指令地址
BX	跳转并切换状态	pc=Rm & 0xffffffe, T=Rm & 1
BLX	带返回的跳转并切换状态	pc=label, T=1 pc=Rm & 0xffffffe, T= Rm&1 lr=BLX 后面的第一条指令地址

地址 label 以一个有符号的相对于 pc 的偏移量保存在指令中, 必须被限制在分支指令的约 32 MB 范围内。T 对应于 cpsr 中的 Thumb 位, 如果指令设置了 T, 那么 ARM 切换到 Thumb 状态。

【例 3.13】 显示一个前向跳转和一个后向跳转。

由于跳转是由地址确定的, 这里不再给出执行前后的状态。这个前向跳转跳过了 3 条指令; 后向跳转建立了一个无限循环。

```

B      forward
ADD    r1,r2,#4
ADD    r0,r6,#2
ADD    r3,r7,#4
forward
SUB    r1,r2,#4

backward
ADD    r1,r2,#4
SUB    r1,r2,#4
ADD    r4,r6,r7
B      backward

```

分支指令用于改变程序的执行流程, 大多数汇编语言通过使用地址标号来隐藏分支指令编码的细节。在例 3.13 中, forward 和 backward 就是地址标号。地址标号放在一行的开始处, 汇编器会记录该行指令的地址, 用于计算跳转的偏移量。

【例 3.14】 带链接的跳转。

带链接的跳转指令和 B 指令相似,不过 BL 指令还要把一个返回地址写到链接寄存器 lr。BL 可用于子程序调用。下面的例子是一个代码片断,使用 BL 指令跳转到一个子程序,再通过拷贝链接寄存器 lr 到 pc 来返回。

```
BL      subroutine      ;跳转到子程序
CMP     r1, #5           ;把 r1 和 5 作比较
MOVEQ   r1, #0           ;if(r1 == 5) then  r1 = 0
...
subroutine
    <子程序代码>
    MOV     pc, lr        ;返回
```

分支切换指令 BX 和带链接的分支切换指令 BLX 是第 3 种类型的分支指令。BX 指令使用一个存储在寄存器 Rm 中的绝对地址,主要用于跳转到 Thumb 代码或从 Thumb 状态返回,在第 4 章中将有描述。cpsr 中的 T 位由分支寄存器的最低位来更新。同样,BLX 指令也是用分支寄存器的最低位来更新 cpsr 中的 T 位,并要把返回地址写入链接寄存器 lr。

3.3 load-store 指令

load-store 指令用于在存储器和处理器寄存器之间传输数据。共有 3 种类型的 load-store 指令:单寄存器传输指令、多寄存器传输指令和交换指令。

3.3.1 单寄存器传送指令

这种指令用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字(32 位)、半字(16 位)和字节。下面是不同的单寄存器传送指令的格式。

语法:

```
<LDR|STR>{<cond>}{B} Rd, addressing1
LDR{<cond>} SB|H|SH Rd, addressing2
STR{<cond>} H Rd, addressing2
```

LDR	把一个字装入一个寄存器	$Rd \leftarrow mem32[address]$
STR	从一个寄存器保存一个字或者一字节	$Rd \rightarrow mem32[address]$
LDRB	把一字节装入一个寄存器	$Rd \leftarrow mem8[address]$
STRB	从一个寄存器保存一字节	$Rd \rightarrow mem8[address]$
LDRH	把一个半字装入寄存器	$Rd \leftarrow mem16[address]$
STRH	从一个寄存器保存一个半字	$Rd \rightarrow mem16[address]$
LDRSB	把一个有符号字节装入寄存器	$Rd \leftarrow signExtend(mem8[address])$
LDRSH	把一个有符号半字装入寄存器	$Rd \leftarrow SignExtend(mem16[address])$

3.3.2 小节中的表 3.5 和表 3.7 将会对 *addressing1* 和 *addressing2* 进行说明。

【例 3.15】 LDR 和 STR 指令可以装载和存储边界对齐的数据。

所谓边界对齐,就是数据边界地址与该数据类型的大小是一致的。例如,LDR 只能从地址为 4 字节整数倍的存储器地址装载 32 位的字 0,4 和 8 等地址。这个例子中从一个以寄存器 r1 内容为地址的存储器中装载数据,然后再写回到原来的地址。

```
1.
; 读取数据到 r0
;
    LDR r0, [r1]      ; = LDR r0, [r1, #0]
;
; 存储数据
;
    STR r0, [r1]      ; = STR r0, [r1, #0]
```

第一条指令从存放在 r1 中的存储器地址处读取一个字,然后存放到 r0 中;第二条指令把 r0 的值写入寄存器 r1 所指向的地址中。从寄存器 r1 的偏移量为 0,寄存器 r1 被称为基地址寄存器。

3.3.2 单寄存器 load-store 指令的寻址方式

ARM 指令集提供了几种存储器寻址的不同方式,这些方式是以下几种变址模式的组合:回写前变址(preindex with writeback)、前变址(preindex)及后变址(postindex)。

表 3.4 列出了几种变址模式。

表 3.4 变址模式

变址模式	数 据	基址寄存器	示 例
回写前变址	mem[base + offset]	基址寄存器加上偏移	LDR r0,[r1,#4]!
前变址	mem[base + offset]	不变	LDR r0,[r1,#4]
后变址	mem[base]	基址寄存器加上偏移	LDR r0,[r1],#4

注：示例部分使用的“!”表示要把计算出的地址回写到基址寄存器。

【例 3.16】 回写型前变址和前变址的区别是：回写型前变址在计算出新的地址后要用新的地址更新基址寄存器的内容，然后再利用新的基址寄存器进行寻址；而前变址方式虽然也利用对基址寄存器的改变值进行寻址，但基址寄存器在操作之后仍然保持原值。后变址和回写型前变址类似，也要更新基址寄存器的内容，但后变址方式先利用基址寄存器的原值进行寻址操作，然后再更新基址寄存器。这两种方式在遍历数组时是很有用的。

PRE

```

r0 = 0x00000000
r1 = 0x00090000
mem32[0x00009000] = 0x01010101
mem32[0x00009004] = 0x02020202

```

```
LDR r0 , [r1, #4]!
```

回写型前变址寻址：

POST(1)

```

r0 = 0x02020202
r1 = 0x00009004

```

```
LDR r0 , [r1; #4]
```

前变址寻址：

POST(2)

```

r0 = 0x02020202
r1 = 0x00009000

```

```
LDR r0 , [r1], #4
```

后变址寻址：

POST(3)

ARM 嵌入式系统开发

```
r0 = 0x01010101
```

```
r1 = 0x00009004
```

例 3.16 说明了在相同的前置条件下,不同的变址方式是如何影响寄存器 r1 中的地址和装载到寄存器 r0 的数据结果的。

对于一条特定的 load-store 指令,其寻址模式依赖于其所属的指令类。表 3.5 中列出了 32 位字(word)和无符号字节(unsigned byte)的 load-store 指令所支持的寻址模式。

表 3.5 单寄存器传输寻址(数据类型是 word 或 unsigned byte)

寻址方式 1(Addressing1)	Addressing1 语法
立即数偏移前变址寻址	$[Rn, \# + / - offset_12]$
寄存器偏移前变址寻址	$[Rn, + / - Rm]$
比例寄存器偏移前变址寻址	$[Rn, + / - Rm, shift_imm]$
立即数偏移回写前变址寻址	$[Rn, \# + / - offset_12]!$
寄存器偏移回写前变址寻址	$[Rn, + / - Rm]!$
比例寄存器偏移回写前变址寻址	$[Rn, + / - Rm, shift \# shift_imm]!$
立即数后变址寻址	$[Rn], + / - offset_12$
寄存器后变址寻址	$[Rn], + / - Rm$
比例寄存器后变址寻址	$[Rn], + / - Rm, shift \# shift_imm$

有符号的偏移量或寄存器用“+/-”表示,表示相对一个基址寄存器 Rn 的正偏移量或负偏移量。基址寄存器是一个指向字节的指针,偏移量表示偏移的字节数。

寻址方式中的立即数说明地址是通过基址寄存器和一个编码在指令中的 12 位偏移量计算而得;寄存器说明地址是通过基址寄存器和另一个特定的寄存器中的内容计算而得;比例寄存器(scaled register)是指使用基址寄存器和一个桶形移位器来计算地址。*

表 3.6 提供了 LDR 指令的不同变体的操作示例。表 3.7 列出了 16 位半字或有符号字节的 load 和 store 指令可以使用的寻址方式。

表 3.6 使用不同寻址方式的 LDR 指令示例

寻址模式	指令	r0 =	r1 +=
回写前变址寻址	LDR r0,[r1,#0x04]!	mem32[r1+0x04]	0x04
	LDR r0,[r1,r2]!	mem32[r1+r2]	r2
	LDR r0,[r1,r2,LSR#0x04]!	mem32[r1+(r2 LSR 0x04)]	(r2 LSR 0x4)

* 由于使用桶形移位器来计算比例,比例因子只能是 2 的倍数。——译者注

续表 3.6

寻址模式	指令	r0 =	r1 +=
前变址寻址	LDR r0,[r1,#0x4]	mem32[r1+0x4]	不变
	LDR r0,[r1,r2]	mem32[r1+r2]	不变
	LDR r0,[r1,-r2,LSR #0x4]	mem32[r1-(r2 LSR #0x4)]	不变
后变址寻址	LDR r0,[r1],#0x4	mem32[r1]	0x4
	LDR r0,[r1],r2	mem32[r1]	r2
	LDR r0,[r1],r2,LSR #0x4	mem32[r1]	(r2 LSR 0x4)

表 3.7 单寄存器 load-store 指令的寻址方式(半字、有符号半字、有符号字节及双字)

Addressing2 方式和变址方法	Addressing2 语法
立即数偏移前变址寻址	[Rn,#+/-offset_8]
寄存器偏移前变址寻址	[Rn,+/-Rm]
立即数偏移回写前变址寻址	[Rn,#+/-offset_8]!
寄存器偏移回写前变址寻址	[Rn,#/-Rm]!
立即数后变址寻址	[Rn],#+/-offset_8
寄存器后变址寻址	[Rn],+/-Rm

这些操作不能使用桶形移位器。其没有 STRSB 或者 STRSH 指令,因为 STRH 可以存储一个有符号或者无符号的半字;同样,STRB 可以存储有符号或者无符号的字节。表 3.8列出了 STRH 指令的各种变体。

表 3.8 STRH 指令的不同变体

寻址模式	指令	结果	R1 +=
回写前变址寻址	STRH r0,[r1,#0x4]!	mem16[r1+0x4]=r0	0x4
	STRH r0,[r1,r2]!	mem16[r1+r2]=r0	r2
前变址寻址	STRH r0,[r1,#0x4]	mem16[r1+0x4]=r0	不变
	STRH r0,[r1,r2]	mem16[r1+r2]=r0	不变
后变址寻址	STRH r0,[r1],#0x4	mem16[r1]=r0	0x4
	STRH r0,[r1],r2	mem16[r1]=r0	r2

3.3.3 多寄存器传送指令

多次装载-存储的 load-store 指令可以用一条指令传送多个寄存器的值到内存,或者从

ARM 嵌入式系统开发

内存取数据到多个寄存器。传输是从一个指向存储器的基地址寄存器 R_n 开始的。多寄存器传送指令在数据块操作、上下文切换、堆栈操作等方面,比单寄存器传送指令会有更高的执行效率。

多寄存器的 load-store 指令会增加中断的延迟,因为 ARM 通常不会打断正在执行的指令去响应中断,而必须等到指令执行完。例如在 ARM7 中,一个多寄存器的 load 指令如果要装载 N 个寄存器,那么指令执行需要 $2+Nt$ 个周期,其中 t 是一次顺序访问存储器所需的周期数。如果一个中断在多寄存器 load-store 指令执行期间产生,那么处理器在多寄存器 load-store 指令执行完后才对中断响应。

编译器,比如 armcc,提供了一个开关来控制一句 load-store 指令可以传送的最大寄存器数目,以限制最大的中断延迟。

语法:

$\langle \text{LDM} | \text{STM} \rangle \{ \langle \text{cond} \rangle \} \langle \text{寻址模式} \rangle R_n \{ ! \}, \langle \text{Registers} \rangle \{ r \}$

LDM	装载多个寄存器	$\{Rd\} * N \leftarrow \text{mem32}[\text{start address} + 4 * N]$ optional R_n updated
STM	保存多个寄存器	$\{Rd\} * N \rightarrow \text{mem32}[\text{start address} + 4 * N]$ optional R_n updated

表 3.9 列出了多寄存器 load-store 指令的不同寻址模式,其中 N 是操作寄存器的个数。

表 3.9 多寄存器传送 load-store 指令的寻址模式

寻址模式	描述	起始地址	结束地址	$R_n!$
IA	执行后增加	R_n	$R_n + 4 * N - 4$	$R_n + 4 * N$
IB	执行前增加	$R_n + 4$	$R_n + 4 * N$	$R_n + 4 * N$
DA	执行后减少	$R_n - 4 * N + 4$	R_n	$R_n - 4 * N$
DB	执行前减少	$R_n - 4 * N$	$R_n - 4$	$R_n - 4 * N$

任何当前寄存器组的子集都可以使用多寄存器 load-store 指令与存储器进行数据交换。基址寄存器 R_n 决定目标或者源地址,可以通过选择使用 R_n 后缀字符“!”来确定 R_n 的值是否随着传送而改变,就像使用回写前变址寻址的单寄存器传送指令一样。

【例 3.17】 寄存器 r_0 作为基地址寄存器 R_n ,并且使用了“!”后缀,表示在指令执行后寄存器将被更新。

注意: 在多寄存器传送指令中,这些寄存器并没有被单独列出来,而是使用了“-”来表示一个寄存器范围。在本例中,这个范围是 $r_1 \sim r_3$ 。如果列出每一个寄存器,则要用逗号分隔,并使用“{ }”把它们括起来。

PRE

```
mem32[0x80018] = 0x03
```

```
mem32[0x80014] = 0x02
```

```
mem32[0x80010] = 0x01
```

```
r0 = 0x00080010
```

```
r1 = 0x00000000
```

```
r2 = 0x00000000
```

```
r3 = 0x00000000
```

```
LDMIA = r0!, {r1 - r3}
```

POST

```
r0 = 0x0008001c
```

```
r1 = 0x00000001
```

```
r2 = 0x00000002
```

```
r3 = 0x00000003
```

图 3.3 是一个图形化的表示。在执行前,基址寄存器 r0 指向存储器地址 0x80010。存储器地址 0x80010,0x80014 及 0x40018 保存的内容分别为 1,2,3。load 指令执行后,r1,r2 及 r3 的值发生改变;最后,基址寄存器 r0 指向了存储器地址 0x8001c,如图 3.4 所示。

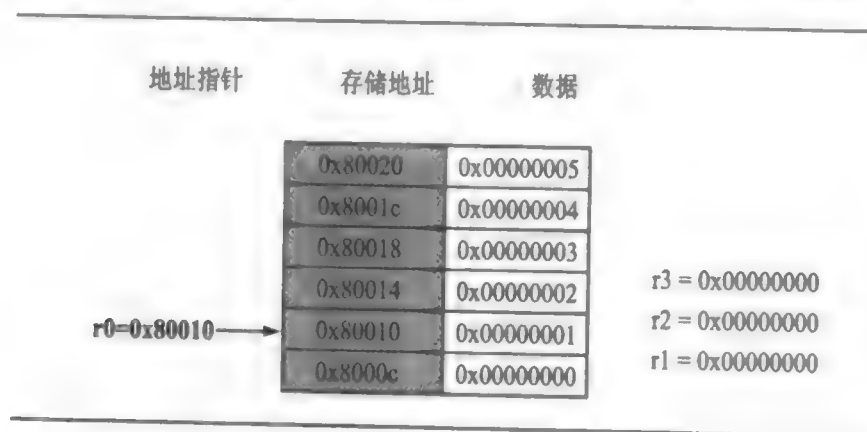


图 3.3 指令执行的前置条件

如果例子中的 LDMIA 指令用 LDMIB 取代,并且是执行前的状态,那么执行后的状态如图 3.4 所示。

在相同的前置条件下,如果把上述 LDMIA 指令换成前增量的 LDMIB 指令,那么寄存器 r0 所指向的第一个字将被忽略,寄存器 r1 将从下一个地址装载数据,如图 3.5 所示。执行后,寄存器 r0 指向最后一个被装载的字。

递减类的多寄存器传送指令(DA 和 DB)是从起始地址递减的,这对于存储器地址与寄

地址指针	存储地址	数据
$r0=0x8001c \rightarrow$	0x80020	0x00000005
	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
	0x80010	0x00000001
	0x8000c	0x00000000

$r3 = 0x00000003$
 $r2 = 0x00000002$
 $r1 = 0x00000001$

图 3.4 LDMIA 指令执行后的情况

地址指针	存储地址	数据
$r0=0x8001c \rightarrow$	0x80020	0x00000005
	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
	0x80010	0x00000001
	0x8000c	0x00000000

$r3 = 0x00000004$
 $r2 = 0x00000003$
 $r1 = 0x00000002$

图 3.5 LDMIB 指令执行后的情况

寄存器编号移动方向相反的传送是很有效的。可以使用递增和递减的多寄存器传送来正向或反向访问数组,或用于堆栈的压入和退出操作。本节稍后的部分将会对此举例说明。

表 3.10 列出了更新基地址的 load-store 指令对。如果使用了表中的一条 store 指令,那么与之配对的 load 指令(相同寄存器数目)将重新装载数据并恢复基地址指针。这对于需要临时保存一组寄存器,然后再恢复它们的场合,是很有用的。

表 3.10 更新基地址的 load-store 指令对

多寄存器 store 指令	多寄存器 load 指令
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

【例 3.18】 显示 STMIB 与 LDMDA 的组合使用。

STMIB 指令把值 7,8,9 存入存储器,然后 $r1 \sim r3$ 被改写,最后使用 LDMDA 恢复了 $r1, r2$ 及 $r3$ 的原始值,并恢复了基地址指针 $r0$ 。

PRE

```
r0 = 0x00009000
```

```
r1 = 0x00000009
```

```
r2 = 0x00000008
```

```
r3 = 0x00000007
```

```
STMIB r0!, {r1 - r3}
```

```
MOV r1, #1      ;改写 r1~r3
```

```
MOV r2, #2
```

```
MOV r3, #3
```

PRE(2)

```
r0 = 0x0000900c
```

```
r1 = 0x00000001
```

```
r2 = 0x00000002
```

```
r3 = 0x00000003
```

```
LDMDA r0!, {r1 - r3}
```

POST

```
r0 = 0x00009000
```

```
r1 = 0x00000009
```

```
r2 = 0x00000008
```

```
r3 = 0x00000007
```

【例 3.19】 使用多寄存器传送指令来完成一个存储器数据块拷贝。
下面的代码从源地址拷贝 32 字节到目标地址。

```
;
```

```
; r9 存放源数据起始地址
```

```
; r10 存放目标起始地址
```

```
; r11 存放源的结束地址
```

```
loop
```

```
LDMLA r9!, {r0 - r7}      ;装载 32 字节并更新 r9 指针
```

```
STMIA r10!, {r0 - r7}     ;存储 32 字节并更新 r10 指针
```

```
CMP    r9,    r11          ;到达结束地址?
```

BNE loop

;不相等跳转

这个程序假设代码执行前已设置好了寄存器 r9、r10 和 r11。寄存器 r9 和 r11 决定了拷贝数据的范围，r10 为数据存储的目标地址。LDMIA 指令装载由 r9 指向的数据到寄存器 r0~r7，同时更新 r9 指向下一个要拷贝的数据块。STMIA 指令拷贝寄存器 r0~r7 的内容到由 r10 指向的目标存储器，同时也更新 r10 指向下一个目标数据块。CMP 和 BNE 指令比较指针 r9 和 r11，检查是否已到达结束地址。如果块拷贝已完成，则程序结束；否则循环将以更新过的 r9 和 r10 继续。BNE 是附条件标志 NE(不相等)的分支指令。

图 3.6 是对上述存储器块拷贝的示意图。理论上讲，这个循环使用 2 条指令就可以传送 32 字节，所以在 33 MHz 的时钟下，可以获得 64 MB/s 的吞吐量。以上数据是在使用快速内存的理想存储系统中得到的。

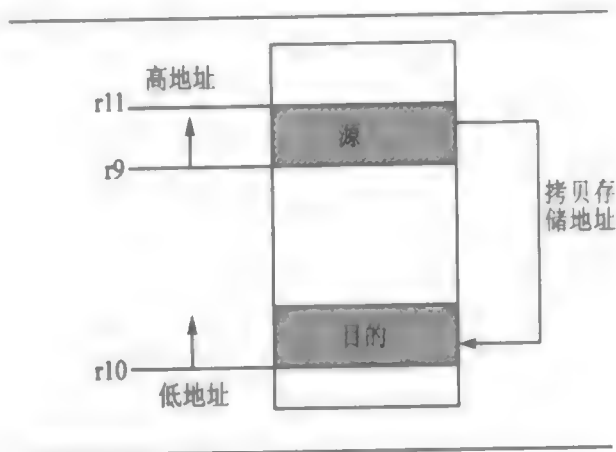


图 3.6 存储器块拷贝示意图

堆栈操作

ARM 体系结构使用多寄存器的 load-store 指令来完成堆栈操作。pop 操作(出栈)使用一条多寄存器的 load 指令，push 操作(入栈)使用一条多寄存器的 store 指令。

在使用一个堆栈的时候，需要确定堆栈在存储器空间中是向上生长还是向下生长的。一个堆栈或者是递增的(ascending, “A”)——向上(高地址空间)生长，或者是递减的(descending, “D”)——向下(低地址空间)生长。

满堆栈(full stack, “F”)是指堆栈指针 sp 指向堆栈的最后一个已使用的地址或满位置(也就是 sp 指向堆栈的最后一个数据项位置)；相反，空堆栈(empty stack, “E”)是指 sp 指向堆栈的第一个没有使用的地址或空位置(也就是 sp 指向堆栈的最后一个数据项的下一个位置)。

有一些多寄存器 load-store 指令的别名支持堆栈操作(见表 3.11)。在 pop 的右边下一列，是与之实际等价的 load 指令。例如，一个递增式满堆栈将由符号 FA 附加在 load 指令——LDMFA，这可以转换成一条 LDMDA 指令。

表 3.11 堆栈操作寻址方式

寻址方式	说 明	pop	-LDM	push	-STM
FA	递增满	LDMFA	LDMDA	STMFA	STMIB
FD	递减满	LDMFD	LDMIA	STMFD	STMDB
EA	递增空	LDMEA	LDMDB	STMEA	STMIA
ED	递减空	LDMED	LDMIB	STMED	STMDA

ARM 制定了 ARM-Thumb 过程调用标准 (ATPCS), 定义了例程如何被调用, 寄存器如何被分配。在 ATPCS 中, 堆栈被定义为递减式满堆栈, 因此 LDMFD 和 STMFD 指令分别用来支持 pop 和 push 功能。

【例 3.20】 STMFD 指令把寄存器内容放入堆栈, 并更新 sp 的值。

图 3.7 显示了在一个递减式满堆栈上的 push 操作, 可以看到堆栈指针的变化并指向堆栈的满位置。

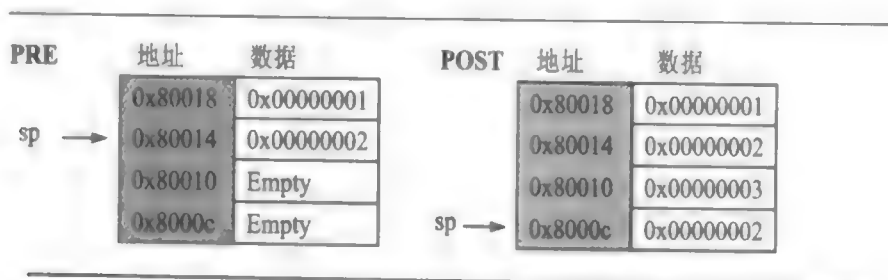


图 3.7 STMFD 指令——递减式满堆栈的 push 操作

PRE

r1 = 0x00000002

r4 = 0x00000003

sp = 0x00000004

STMFD sp!, {r1, r4}

POST

r1 = 0x00000002

r4 = 0x00000003

sp = 0x0008000c

【例 3.21】 显示在一个递减式空堆栈上, 使用 STMED 指令完成的一个 push 操作 (见图 3.8)。STMED 指令把寄存器内容压栈, 但 sp 指向了下一个空位置。

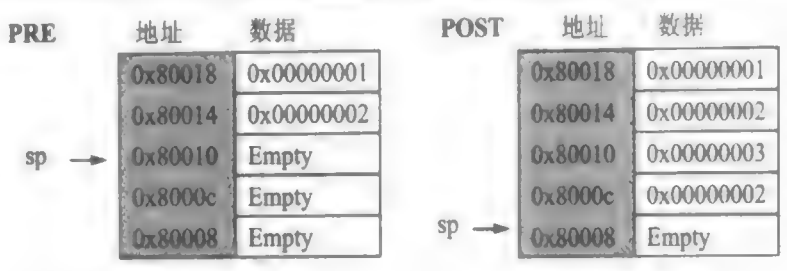


图 3.8 STMED 指令——递减式空堆栈的 push 操作

PRE

r1 = 0x00000002

r4 = 0x00000003

sp = 0x00080010

STMED sp!, {r1, r4}

POST

r1 = 0x00000002

r4 = 0x00000003

sp = 0x00080008

若要检查一个堆栈,须关注堆栈的 3 个属性:堆栈基址、堆栈指针及堆栈限制。堆栈基址是堆栈在存储器中的起始地址;堆栈指针初始时指向堆栈基址单元,随着数据压栈,堆栈指针连续移动并始终指向栈顶;如果堆栈指针超出了堆栈限制,就会发生堆栈溢出错误。下面的一小段代码用于检测递减式堆栈的溢出错误:

```
SUB sp, sp, #size
CMP sp, r10
BLL0 _stack_overflow ;条件
```

ATPCS 把寄存器 r10 定义为堆栈限制或 sl(stack limit)。这是一个可选的操作,因为堆栈检查只有在堆栈检查使能的时候才可使用。BLL0 指令是一个加了条件助记符 L0 的带链接的分支指令。如果在执行一个 push 操作后,sp 小于 r10,就发生了堆栈溢出错误。如果堆栈指针在执行 pop 操作后,超出了堆栈基址,那么就产生了堆栈下溢(stack under-flow)错误。

3.3.4 交换指令

交换指令是 load-store 指令的一种特例,它把一个存储器单元的内容与寄存器内容相交换。交换指令是一个原子操作(atomic operation)——在连续的总线操作中读/写一个存储单元,在操作期间阻止其它任何指令对该存储单元的读/写。

语法:

SWP {B} {<cond>} Rd, Rm, [Rn]

SWP	字交换	tmp=mem32[Rn] mem32[Rn]=Rm Rd= tmp
SWPB	字节交换	tmp=mem8[Rn] mem8[Rn]=Rm Rd= tmp

交换指令在执行期间不能被其它任何指令或其它任何总线访问打断,在此期间系统“占据总线(holds the bus)”,直至交换完成。

【例 3.22】 使用交换指令把一个存储器单元的内容放到寄存器 r0,然后把 r1 的内容存储到该内存单元中去。

PRE

mem32[0x9000] = 0x12345678

r0 = 0x00000000

r1 = 0x11112222

r2 = 0x00009000

SWP r0, r1, [r2]

POST

mem32[0x9000] = 0x11112222

r0 = 0x12345678

r1 = 0x11112222

r2 = 0x00009000

交换指令多用于实现操作系统中的信号量和互斥操作。从指令的语法可以看到,这条

ARM 嵌入式系统开发

指令可以有一字节修饰符 B, 所以交换指令可以有字交换和字节交换两种形式。

【例 3.23】 显示使用 SWP 指令来保证数据交换期间不被其它任务改写, 是一个信号量操作的简单实现。SWP 指令“占据总线”, 直至交换完成。

```
spin
    MOV r1, = semaphore
    MOV r2, #1
    SWP r3, r2, [r1] ;hold the bus until complete
    CMP r3, #1
    BEQ spin
```

例子中, 由信号量(semaphore)指向的地址单元中或者是 0, 或者是 1。如果信号量为 1, 则说明该服务正被另一个过程使用, 程序将继续循环, 直至该服务被释放——信号量地址位置的值变为 0。

3.4 软件中断指令

软件中断指令(SWI)可以产生一个软件中断异常, 这为应用程序调用系统例程提供了一种机制。

语法:

SWI {<cond>} SWI_number

SWI	软件中断	$lr_svc = \text{SWI 指令后面的指令地址}$ $spsr_svc = cpsr$ $pc = \text{vectors} + 0x8$ $cpsr \text{ 模式} = \text{SVC}$ $cpsr I = 1 \text{ (屏蔽 IRQ 中断)}$
-----	------	---

处理器执行 SWI 指令时, 设置程序计数器 pc 为向量表的 0x8 偏移处, 同时强制切换处理器模式到 SVC 模式, 以便操作系统例程可以在特权模式下被调用。

每个 SWI 指令有一个关联的 SWI 号(number), 用于表示一个特定的功能调用或特性。

【例 3.24】 一个 ARM 工具箱中用于调试 SWI 的例子, 是一个 SWI 号为 0x123456 的 SWI 调用。通常 SWI 指令是在用户模式下执行的。

PRE

```

cpsr = nzcVqift_USER
pc = 0x00008000
lr = 0x003ffffff ;lr = r4
r0 = 0x12

```

```

0x00008000    SWI    0x123456

```

POST

```

cpsr = nzcVqift_SVC
spsr = nzcVqift_USER
pc = 0x00000008
lr = 0x00008004
r0 = 0x12

```

SWI 用于调用操作系统的例程,通常需要传递一些参数,这可以通过使用寄存器来完成。在上面的例子中,r0 用于传递参数 0x12,返回值也通过寄存器来传递。

处理软件中断调用的代码段称为中断处理程序(SWI Handler)。中断处理程序通过执行指令的地址获取软件中断号,指令地址是从 lr 计算出来的。

SWI 号由下式决定:

$$\text{SWI_Number} = \langle \text{SWI instruction} \rangle \text{ AND NOT}(0\text{xff}000000)$$

其中 SWI instruction 就是实际处理器执行的 32 位 SWI 指令。

【例 3.25】 SWI 处理程序开始部分的实现。

这个片段代码计算了正被调用的 SWI 号,并把它放进 r10。从这个例子中可以看到,load 指令先拷贝整个 SWI 指令到寄存器 r10,然后使用 BIC 屏蔽了指令的高 8 位,获取 SWI 号。这里假定 SWI 是在 ARM 状态下被调用的。

```

SWI_handler
{
    ; 保存寄存器 r0~r12 和 lr
    ;
    STMFD sp!, {r0 - r12,lr}

    ;read the SWI instruction
    LDR r10, [lr, #-4]

```

```

;mask off top 8 bits
BIC r10, r10, #0xff000000

;r10 - contains the SWI number
BL    service_routine

;return from SWI handler
LDMFD sp!, {r0 - r12, pc}~

```

然后,在寄存器 r10 中的值(SWI 号)将被 SWI 处理程序用于调用相关的 SWI 服务子程序。

3.5 程序状态寄存器指令

ARM 指令集提供了 2 条指令,可直接控制程序状态寄存器(psr)。MRS 指令用于把 cpsr 或者 spsr 的值传送到一个寄存器;MSR 与之相反,把一个寄存器的内容传送到 cpsr 或者 spsr。这 2 条指令结合,可用于对 cpsr 和 spsr 进行读/写操作。

指令语法:

```

MRS {<cond>} Rd, <cpsr|spsr>
MSR <<cond>> <cpsr|spsr>_<fields>, Rm
MSR <<cond>> <cpsr|spsr>_<fields>, #immediate

```

MRS	把程序状态寄存器的值送到一个通用寄存器	Rd = spr
MSR	把通用寄存器的值送到程序状态寄存器	psr[field] = Rm
MSR	把一个立即数送到程序状态寄存器	psr[field] = immediate

在指令语法中可看到一个称为 fields(域)的项,它可以是控制(c)、扩展(x)、状态(s)及标志(f)的任意组合。这些域及其在程序状态寄存器中的特定字节区域,如图 3.9 所示。

图 3.9 中,控制域控制中断屏蔽、Thumb 状态和处理器模式。例 3.26 将说明如何通过清除 I 屏蔽位来使能 IRQ 中断。这个例子使用了 MRS 和 MSR 指令来读/写 cpsr。

【例 3.26】 MSR 指令先把 cpsr 的值复制到 r1;然后使用 BIC 清除 r1 的位 7;再使用 MRS 把 r1 的值复制到 cpsr,使能 IRQ 中断。

从这个例子可以看到代码如何保护 cpsr 中的其它设置位,而只修改控制域的 I 位。

PRE

```
cpsr = nzcvcifT_SVC
```

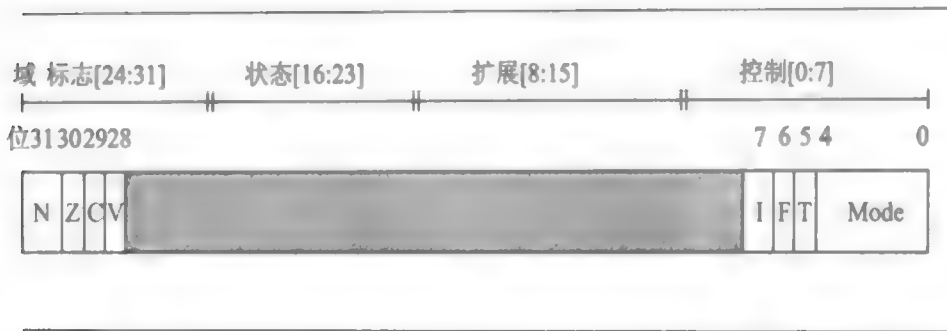


图 3.9 psr 字节域定义

```
MRS r1, cpsr
BIC r1, r1, #0x80 ; 0b01000000
MSR cpsr_c, r1
```

```
POST
cpsr = nzcvqiFt_SVC
```

这个例子是在 SVC 模式下执行的。在用户模式下可以读取 cpsr,但只能更改条件标志域 f。

3.5.1 协处理器指令

协处理器指令用于扩展指令集。协处理器指令既可用于提供附加的计算能力,又可用于控制包括 cache 和内存管理的存储子系统。协处理器指令包括数据处理、寄存器传输及内存传输指令。这里只对协处理器指令进行简要说明,因为协处理器指令和具体的协处理器相关。

注意: 协处理器指令只用于带有协处理器的 ARM 核。

语法:

```
CDP {<cond>} cp, opcode1, Cd, Cn {, opcode2}
<MRC|MCR> {<cond>} cp, opcode1, Rd, Cn, Cm{, opcode2}
<LDC|STC>{<cond>} cp, Cd ,addressing
```

CDP	协处理器数据处理——在协处理器内部执行一个数据处理操作
MRC MCR	协处理器寄存器传输——把数据送入或取出协处理器寄存器
LDC STC	协处理器内存传输——从协处理器装载/存储一个内存数据块

在协处理器指令语法中, *cp* 域代表协处理器的编号, 为 *p0*~*p15*。*opcode* 域描述要在协处理器中执行的操作。*Cn*, *Cm* 及 *Cd* 描述在协处理器中的寄存器。协处理器的操作和寄存器依赖于具体使用的协处理器。协处理器 15(CP15)是为系统控制预留的, 如内存管理、写缓冲控制、cache 控制及寄存器识别等。

【例 3.27】 把一个 CP15 寄存器拷贝到一个通用寄存器。

;把协处理器 CP15 寄存器 *c0* 的内容拷贝到 *r10*

```
MRC p15,0,r10,c0,c0,0
```

这里 CP15 寄存器 *c0* 中包含处理器标识(processor id number), 其内容被拷贝到通用寄存器 *r10*。

3.5.2 协处理器 15(CP15)指令语法

CP15 可以配置处理器核, 并有一组专用的寄存器用于存储配置信息, 如例 3.27 所述。通过写一个值到一个寄存器来设置一种配置属性——比如, 打开 cache。

CP15 被称为系统控制协处理器。MRC 和 MCR 指令用于读/写 CP15, 在语法中 *Rd* 是内核目标寄存器, *Cn* 是主寄存器, *Cm* 是辅寄存器, *opcode2* 是辅寄存器修饰符。辅寄存器通常也被称为“扩展寄存器”。

下面这个例子, 把 CP15 的控制寄存器 *c1* 写到内核寄存器 *r1* 中:

```
MRC p15,0,r1,c1,c0,0
```

为了便于对 CP15 的配置寄存器进行说明, 使用下面的缩写格式:

CP15;cX;cY;Z

第 1 部分, CP15 定义它是协处理器 15; 冒号后的第 2 部分是主寄存器, *X* 是一个 0~15 的值; 第 3 部分是辅寄存器或扩展寄存器, *Y* 可以是一个 0~15 的值; 最后一项是 *opcode2*, *opcode2* 可以是一个 0~7 的值。

有些操作可能用到一个非零值(*w*)的 *opcode1*, 这种格式表示为:

CP15:w;cX;cY;Z

3.6 常量的装载

读者可能已经注意到,ARM 指令不用于把一个 32 位常量装入寄存器。因为 ARM 指令本身是 32 位的,所以指令中不可能再定义一个普通的 32 位常量。

为了便于编程,ARM 增加了 2 条伪指令,用于把一个 32 位的常量送入寄存器。
语法:

```
LDR    Rd,    =constant
ADR    Rd,    label
```

LDR	常量装载伪指令	Rd=32 位常量
ADR	地址装载伪指令	Rd=32 位相对地址

第 1 条伪指令用于把一个 32 位常量送入寄存器,它的具体执行可以是任何可实现该功能的指令。如果要执行的操作不能用其它指令来编码,那么 LDR 就执行存储器读。第 2 条伪指令把一个相对地址写入寄存器中,它将会使用一个包括 pc 相对地址的表达式进行编码。

【例 3.28】 显示使用 LDR 指令把一个 32 位常量 0xff00ffff 装入寄存器 r0。

```
LDR r0, [pc, #constant_number - 8 - {PC}]
...
constant_number
DCD    0xff00ffff
```

这个例子使用了存储器访问来装载常量,这对于时间要求苛刻的程序来说代价是较高的。例 3.29 使用另一种方法来实现同样的功能——使用 MVN 指令。

【例 3.29】 使用 MVN 指令装载常量 0xff00ffff 到寄存器 r0。

```
PRE    <无>
        MVN    r0, #0x00ff0000
POST
        r0 = 0xff00ffff
```

表 3.12 列出两条伪指令在执行不同常量装载时的不同转换。第 1 条伪指令使用了一条简单的 MOV 指令;第 2 条伪指令使用了一个 pc 相对地址的 load,推荐使用这种伪指令来装载常量,让编译器或汇编器来选择实际的转换指令。使用反汇编的方法可以看到编译器或汇编器在常量装载时具体选择的指令。

表 3.12 LDR 伪指令转换

伪指令	实际执行的指令
LDR r0, =0xff	MOV r0, #0xff
LDR r0, =0x55555555	LDR r0, [pc, #offset_12]

从表 3.12 中可以看出,可以避免存储器访问而使用其它的指令来达到同样的效果,但这依赖于所要装载的常量的内容。好的编译器和汇编器会使用灵活的方法来避免从存储器装载常量,这些工具有一些算法来寻找指令所需要产生的最理想的数,并最大限度地使用桶形移位器,用最佳的指令完成操作。如果通过这些方法不能产生所需的常量,那么就使用访问寄存器的方法解决。LDR 伪指令的执行或者是一条 MOV 指令,或者是一条 MVN 指令来产生一个值(如果可能),或者是一条 LDR 指令从一个 pc 相对的存储器地址中读取数据(嵌在存储器代码中的数据区)。

另外一条比较有用的伪指令是 ADR,或者称为一个相对地址指令。它可以把一个指定标号的地址,使用一个 pc 相对地址的加或减放入寄存器 Rd。

3.7 ARMv5E 扩展

ARMv5E 扩展提供了许多新的指令(参见表 3.13)。其中最重要的指令是 16 位数据的有符号乘累加指令。这些指令在许多 ARMv5E 的实现上都可以在一个周期内完成。

ARMv5E 在操作 16 位数据时提供了更好的性能和更大的灵活性,这对于诸如 16 位数字音频处理等许多应用是很重要的。

表 3.13 ARMv5E 新增加的指令

指 令	说 明
CLZ{<cond>} Rd,Rm	零计数
QADD{<cond>} Rd,Rm,Rn	有符号 32 位饱和加法
QDADD{<cond>} Rd,Rm,Rn	有符号双 32 位饱和加法
QDSUB{<cond>} Rd,Rm,Rn	有符号双 32 位饱和减法
QSUB{<cond>} Rd,Rm,Rn	有符号 32 位饱和减法
SMLAxy{<cond>} Rd,Rm,Rs,Rn	有符号 32 位乘累加(1)
SMLALxy{<cond>} RdLo,RdHi,Rm,Rs	有符号 64 位乘累加
SMLAWy{<cond>} Rd,Rm,Rs,Rn	有符号 32 位乘累加(2)
SMULxy{<cond>} Rd,Rm,Rs	有符号乘法(1)
SMULWy{<cond>} Rd,Rm,Rs	有符号乘法(2)

正数 0x7fffffff。这就可以避免使用额外的代码来检查可能的溢出。表 3.14 列出了 ARMv5E 的所有饱和指令。

表 3.14 饱和指令

指 令	饱和计算
QADD	$Rd = Rn + Rm$
QDADD	$Rd = Rn + (Rm * 2)$
QSUB	$Rd = Rn - Rm$
QDSUB	$Rd = Rn - (Rm * 2)$

【例 3.32】 使用 QADD 指令完成例 3.31 同样的加法。

PRE

```
cpsr = nzcvtFt_SVC
```

```
r0 = 0x00000000
```

```
r1 = 0x70000000 (正数)
```

```
r2 = 0x7fffffff (正数)
```

```
QADD r0, r1, r2
```

POST

```
cpsr = nvcvtFt_SVC
```

```
r0 = 0x7fffffff
```

可以看到,在寄存器 r0 中保存了饱和数,同时 Q 位(cpsr 中的位 27)被设置。Q 标志说明饱和发生,它将一直保持,直至程序显式地清除。

3.7.3 ARMv5E 乘法指令

表 3.15 列出了所有 ARMv5E 的乘法指令。其中, x 和 y 分别用于选择一个 32 位寄存器中的 2 个 16 位哪个作为第一操作数,哪个作为第二操作数。这个域设置为 T,表示使用高 16 位;设置为 B,表示使用低 16 位。对于 32 位结果的乘累加操作, Q 标志位指示累加是否溢出了有符号 32 位值。

表 3.15 有符号乘法和乘累加指令

指 令	有符号乘法[累加]	有符号 结果	Q 标 志位	计算操作
SMLAxy	$(16\text{-bit} * 16\text{-bit}) + 32\text{-bit}$	32 位	更 新	$Rd = (Rm, x * Rs, y) + Rn$
SMLALxy	$(16\text{-bit} * 16\text{-bit}) + 64\text{-bit}$	64 位	无影响	$[RdHi, RdLo] += Rm, x * Rs, y$
SMLAWy	$((32\text{-bit} * 16\text{-bit}) \gg 16) + 32\text{-bit}$	32 位	更 新	$Rd = ((Rm * Rs, y) \gg 16) + Rn$
SMULxy	$(16\text{-bit} * 16\text{-bit})$	32 位	无影响	$Rd = Rm, x * Rs, y$
SMULWy	$((32\text{-bit} * 16\text{-bit}) \gg 16)$	32 位	无影响	$Rd = (Rm * Rs, y) \gg 16$

【例 3.33】 显示乘法指令的使用方法。
这里使用有符号乘累加指令 SMLATB。

PRE

```
r1 = 0x20000001
r2 = 0x20000001
r3 = 0x00000004
```

```
SMLATB r4, r1, r2, r3
```

POST

```
r4 = 0x00002004
```

指令把寄存器 r1 的高 16 位乘以寄存器 r2 的低 16 位,结果累加到寄存器 r3,并把它写到目标寄存器 r4 中。

3.8 条件执行

大多数 ARM 指令是可条件执行的,可指定指令只有当条件代码标志与给定的条件匹配时才执行。通过使用条件执行,可以改善性能和提高代码密度。

条件码是跟在指令助记符后面的 2 个字母,缺省条件是 AL(always execute),即无条件执行。

条件执行减少了分支指令的数目,相应地减少了指令流水线的排空次数,从而改善了执行代码的性能。条件执行主要依赖于两部分:条件码和条件标志。条件码位于指令中,条件标志位于 cpsr 中。

ARM 嵌入式系统开发

【例 3.34】 使用一个带有 EQ 条件码的 ADD 指令,指令只有在 cpsr 的 zero 位为 1 时才被执行。

```
;r0 = r1 + r2  if zero flag is set
ADDEQ r0, r1, r2
```

只有带 S 后缀的比较指令和数据处理指令才会更新 cpsr 中的条件标志。

【例 3.35】 为了说明条件执行的优点,通过这个例子中的一个简单的 C 代码片段,比较使用条件执行指令与不使用条件执行指令时的汇编代码情况。

```
while (a!= b)
{
    if (a>b) a -= b; else b -= a;
}
```

这里使用寄存器 r1 代表 a,寄存器 r2 代表 b。下面的代码是在 ARM 汇编器编写的同样算法(求最大公约数)的汇编代码,其中只使用了可条件执行的分支指令,其它指令都不是条件执行的。

```
        ;最大公约数算法
gcd
        CMP    r1, r2
        BEQ    complete
        BLT    lessthan
        SUB    r1, r1,r2
        B      gcd
lessthan
        SUB    r2, r2,r1
        B      gcd
complete
...
```

现在与全部使用条件执行的代码来比较。可见,使用条件执行可大幅度地减少指令数目。

```
gcd
        CMP    r1, r2
        SUBGT   r1, r1, r2
        SUBLT   r2, r2, r1
        BNE     gcd
```

3.9 总 结

本章讲述了 ARM 指令集。所有 ARM 指令都是 32 位的。算术指令、逻辑指令、比较指令及 MOVE 指令可使用内嵌的桶形移位器,以便在第 2 个寄存器 Rm 进入 ALU 之前,对它进行预处理。

ARM 指令集有 3 种类型的 load-store 指令:单寄存器的 load-store 指令、多寄存器的 load-store 指令及交换指令。多寄存器 load-store 指令为堆栈的 push-pop 操作提供了一种有效的实现方式。ARM-Thumb 过程调用标准(ATPCS)把堆栈定义为递减式满堆栈。

软件中断指令产生一个软件中断,使处理器进入 SVC 模式,即这条指令调用了特权操作系统例程。程序状态寄存器指令可对 cpsr 和 spsr 进行读/写。ARM 中还有专门用于优化 32 位常量装载操作的特殊伪指令。

ARMv5E 扩展引入了零计数、饱和运算及增强乘法指令。零计数指令可计算一个二进制数的前导零的个数;饱和运算用来处理超过 32 位整数值的算法计算;增强乘法指令为 16 位乘法提供了更大的灵活性。

大多数的 ARM 指令可被条件执行,对一个特定的算法,使用条件指令可大幅度减少所需的指令数目。

第 4 章

Thumb 指令集

- Thumb 寄存器的使用
- ARM-Thumb 交互
- 其它分支指令
- 数据处理指令
- 单寄存器 load-store 指令
- 多寄存器 load-store 指令
- 堆栈指令
- 软件中断指令
- 总 结

本章介绍 Thumb 指令集。Thumb 把 32 位 ARM 指令集的一个子集进行编码,成为一个 16 位的指令集。在 16 位外部数据总线宽度下,在 ARM 处理器上使用 Thumb 指令的性能要比使用 ARM 指令的性能更好;而在 32 位外部数据总线宽度下,使用 Thumb 指令的性能要比使用 ARM 指令的性能差。因此,Thumb 指令多用于存储器受限的一些系统中。

相对于 ARM 指令集,使用 Thumb 指令集可获得更高的代码密度——一个可执行的程序在内存中所占的空间。在存储器受限的嵌入式系统中,比如移动电话、PDA 等,代码密度是非常重要的;同时,成本压力也会限制存储器的大小、数据宽度和速度。

平均而言,对于同一个程序,使用 Thumb 指令实现所需的存储空间,要比等效的 ARM 指令实现少 30% 左右。图 4.1 显示了对于实现同样的除法运算,使用 ARM 指令和使用 Thumb 指令的汇编代码。虽然 Thumb 指令的实现使用了更多的指令,但是它所占的总的存储空间却比较小。代码密度是 Thumb 指令集的一个主要优势。由于 Thumb 指令集的设计是面向编译器的,而不是针对手写汇编的,所以推荐使用高级语言如 C 或者 C++ 语言来编程,然后用编译器生成 Thumb 指令的目标代码。

ARM 代码	Thumb 代码
ARM 除法	Thumb 除法
; 输入: r0(值), r1 (除数)	; 输入: r0(值), r1 (除数)
; 输出: r2 (模), r3 (商)	; 输出: r2 (模), r3 (商)
MOV r3, #0	MOV r3, #0
loop	loop
SUBS r0, r0, r1	ADD r3, #1
ADDGE r3, r3, #1	SUB r0, r1
BGE loop	BGE loop
ADD r2, r0, r1	SUB r3, #1
	ADD r2, r0, r1
5×4=20 bytes	6×2=12 bytes

图 4.1 代码密度

每一条 Thumb 指令都和一条 32 位的 ARM 指令相关,图 4.2 显示了一条 Thumb 加法指令 ADD 译码成等效的 ARM 加法指令。

表 4.1 给出了在 ARMv5TE 架构下的 THUMBv2 中所有的 Thumb 指令。在 Thumb ISA 中,只有分支指令可被条件执行;同时由于 16 位空间的限制,桶形移位操作如 ASR, LSL, LSR 和 ROR,也变成单独的指令。

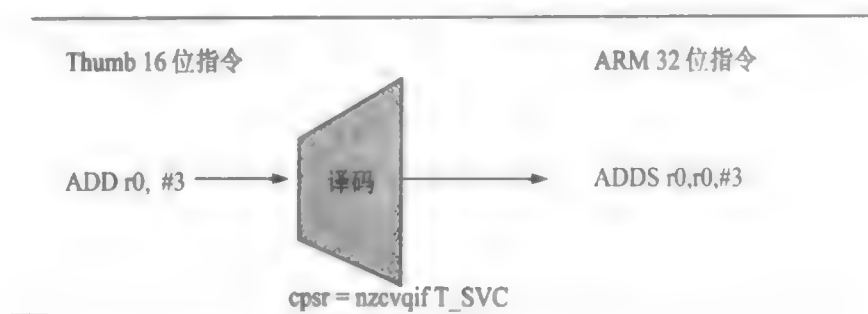


图 4.2 Thumb 指令译码

表 4.1 Thumb 指令集

助记符	Thumb ISA	描述
ADC	v1	带进位的 32 位加
ADD	v1	32 位加
AND	v1	32 位逻辑位与
ASR	v1	算术右移
B	v1	跳转指令
BIC	v1	32 位逻辑位清除 (AND NOT)
BKPT	v2	断点指令
BL	v1	带链接相对跳转指令
BLX	v2	带交换的跳转指令
CMN	v1	32 位相反数比较
CMP	v1	32 位比较
EOR	v1	32 位逻辑异或
LDM	v1	从内存装载多个 32 位字到 ARM 寄存器
LDR	v1	从一个内存虚拟地址转载一个单一的值
LSL	v1	逻辑左移
LSR	v1	逻辑右移
MOV	v1	把一个 32 位值送入寄存器
MUL	v1	32 位乘法
MVN	v1	把一个 32 位值逻辑取反后送入寄存器
NEG	v1	取反
ORR	v1	32 位逻辑或
POP	v1	从堆栈中取多个值到寄存器
PUSH	v1	多寄存器入栈

续表 4.1

助记符	Thumb ISA	描 述
ROR	v1	32 位循环右移
SBC	v1	带进位的 32 位减法
STM	v1	把多个 32 位寄存器存放到内存
STR	v1	把一个寄存器的值存放到一个内存的虚拟地址
SUB	v1	32 位减法
SWI	v1	软中断
TST	v1	32 位位测试

4.1 Thumb 寄存器的使用

在 Thumb 状态下,不能直接访问所有的寄存器,只有寄存器 $r0 \sim r7$ 是可以被任意访问的,如表 4.2 所列。寄存器 $r8 \sim r12$ 只能通过 MOV, ADD 或 CMP 指令来访问。CMP 指令和所有操作 $r0 \sim r7$ 的数据处理指令都会影响 cpsr 中的条件标志。

表 4.2 Thumb 寄存器的使用

寄存器	访 问
$r0 \sim r7$	完全访问
$r8 \sim r12$	只能通过 MOV, ADD 及 CMP 访问
$r13$ sp	限制访问
$r14$ lr	限制访问
$r15$ pc	限制访问
cpsr	只能间接访问
spsr	不能访问

从表 4.1 和表 4.2 可以看出,不能直接访问 cpsr 和 spsr。换句话说,没有与 MSR 和 MRS 等价的 Thumb 指令。

为了改变 cpsr 或 spsr 的值,必须切换到 ARM 状态,使用 MSR 和 MRS 来实现。同样,在 Thumb 状态下没有协处理器指令,要访问协处理器来配置 cache 和进行内存管理,也必须在 ARM 状态下。

4.2 ARM-Thumb 交互

ARM-Thumb 交互是指对汇编语言和 C/C++ 语言的 ARM 和 Thumb 代码进行连接的方法,它进行两种状态(ARM 和 Thumb)间的转换。在进行这种转换时,有时须使用额外的代码,这些代码被称为胶合(*veneer*)。ATPCS 定义了 ARM 和 Thumb 过程调用的标准。

从一个 ARM 例程调用一个 Thumb 例程,内核必须切换状态。状态的变化由 cpsr 中的 T 位来显示。在跳转到一个例程时,BX 和 BLX 分支指令可用于 ARM 和 Thumb 状态的切换。BX lr 指令从一个例程返回,如果需要,也可以进行状态切换。

BLX 指令在 ARMv5T 中引入。在 ARMv4T 核中,连接器在子程序调用时,使用胶合来完成状态的切换。连接器不是直接调用例程,而是通过调用胶合,由胶合使用 BX 指令来切换到 Thumb 状态。

有 2 个版本的 BX 和 BLX 指令:ARM 指令和等效的 Thumb 指令。ARM BX 指令只有当 Rn 中地址的最低位为 1 时,才进入 Thumb 状态;否则进入 ARM 状态。Thumb BX 指令以同样的方式工作。

语法:

```
BX      Rm
BLX     Rm | label
```

BX	Thumb 版本分支切换	pc = Rn & 0xffffffe T = Rn[0]
BLX	Thumb 版本带链接的分支切换	lr = BLX 后面的指令地址 + 1 pc = label, T = 0 pc = Rm & 0xffffffe, T = Rm[0]

与 ARM 版本的 BX 指令不同,Thumb BX 指令不能被条件执行。

【例 4.1】 显示使用 ARM BX 和 Thumb BX 的一个小代码段。

可以看到,进入 Thumb 的分支地址的最低位是置 1 的,这将置位 cpsr 中的 T 位而进入 Thumb 状态。

使用 BX 指令时,返回地址不是自动保留的,因而在跳转指令调用前,通过显式地使用 MOV 指令来设置返回地址。


```

;ARM 代码
CODE32                                ;字对齐
LDR    r0, = thumbCode + 1           ;+1 进入 Thumb 状态
MOV     lr, pc                        ;设置返回地址
BX      r0                            ;返回 Thumb 状态
;继续其它的代码

;Thumb 代码
CODE16                                ;半字对齐
thumbCode
ADD     r1, #1
BX      lr                            ;返回 ARM 状态

```

如果没有通过位 0 的强制置 1 来进行状态转换,那么分支切换指令也可以用作一个绝对跳转指令。

```

;    address(thumbCode) = 0x00010000
;    cpsr = nzcvcIfT_SVC
;    r0 = 0x00000000
0x00009000    LDR r0,    = thumbCode + 1
;    cpsr = nzcvcIfT_SVC
;    r0 = 0x00010001
0x00009008    BX r0
;    cpsr = nzcvcIfT_SVC
;    r0 = 0x00010001
;    pc = 0x00010000

```

可以看到, r0 的最低位用于置位 cpsr 中的 T 位, cpsr 从执行 BX 指令前的 IfT 变为执行后的 IFT, pc 指向到 Thumb 例程的起始地址。

【例 4.2】 使用 BLX 指令代替 BX 指令简化了 Thumb 例程的调用, 因为 BLX 指令在链接寄存器 lr 中自动设置了返回地址。

```

CODE32
LDR    r0, = thumbRoutine + 1        ;进入 Thumb 状态
BLX     r0                            ;跳到 Thumb 代码
;其它代码
CODE16
ThumbRoutine
ADD     r1, #1
BX      r14                            ;返回到 ARM 状态

```

4.3 其它分支指令

有 2 个标准分支指令的变体,即 B 指令:第 1 个变体与 ARM 版本指令相似,可条件执行,跳转被限制在有符号 8 位立即数所表示的范围内,或者是 $-256 \sim 254$ 字节;第 2 个变体不可条件执行(没有条件码部分),但扩展了有效跳转范围——有符号的 11 位立即数表示的范围,或 $-2\,048 \sim +2\,046$ 字节。

条件分支指令是 Thumb 指令中惟一可以条件执行的指令。

语法:

B<cond> label
B label
BL label

B	分支指令	pc = label
BL	带链接的分支指令	pc = label lr = BL 后的指令地址 + 1

BL 指令不可条件执行,可以在大约 ± 4 MB 的范围内跳转,因为 BL(和 BLX)指令被转换成一对 16 位的 Thumb 指令,因而,上述跳转范围是合理的。这对指令中的第 1 条包含跳转偏移量的高位部分,第 2 条包含其低位部分,这些指令必须成对使用。

这里列出了从 BL 子程序调用返回的不同指令:

MOV pc,lr
BX lr
POP {pc}

POP 堆栈操作指令将在 4.7 节中详细讨论。

4.4 数据处理指令

数据处理指令可以操作寄存器中的数据,包括 MOV 指令、算术指令、移位指令、逻辑指令、比较指令和乘法指令。Thumb 数据处理指令是 ARM 数据处理指令的一个子集。

语法:

(ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB) Rd,Rn

(ADD|ASR|LSL|LSR|ROR|SUB) Rd, Rn # immediate

(ADD|MOV|SUB) Rd, # immediate

(ADD|SUB) Rd,Rn,Rn

ADD Rd, pc, # immediate

ADD Rd, sp, # immediate

(ADD|SUB) sp, # immediate

(ASR|LSL|LSR|ROR) Rd, Rs

(CMN|CMP|TST) Rn, Rn

CMP Rn, # immediate

MOV Rd, Rn

ADC	带进位的 32 位加	$Rd = Rd + Rn + C\text{ flag}$
ADD	32 位加	$Rd = Rn + \text{immediate}$ $Rd = Rd + \text{immediate}$ $Rd = Rd + Rn$ $Rd = Rd + Rn$ $Rd = (pc \& 0xffffffe) + (\text{immediate} \ll 2)$ $Rd = sp + (\text{immediate} \ll 2)$ $Rd = sp + (\text{immediate} \ll 2)$
AND	32 位逻辑“与”	$Rd = Rd \& Rn$
ASR	算术右移	$Rd = Rn \gg \text{immediate}$ $C\text{ flag} = Rn[\text{immediate}-1]$ $Rd = Rd \gg Rs, C\text{ flag} = Rd[Rs-1]$
BIC	32 位逻辑位清零	$Rd = Rd \text{ AND NOT } (Rn)$
CMN	32 位取负比较	$Rn + Rn \quad \text{set flags}$
CMP	32 位整数比较	$Rn - \text{immediate} \text{ set flags}$ $Rn - Rn \quad \text{set flags}$
EOR	32 位逻辑“异或”	$Rd = Rd \text{ EOR } Rn$
LSL	逻辑左移	$Rd = Rn \ll \text{immediate}$ $C\text{ flag} = Rn[32 - \text{immediate}]$ $Rd = Rd \ll Rs, C\text{ flag} = Rd[32 - Rs]$
LSR	逻辑右移	$Rd = Rn \gg \text{immediate}$ $C\text{ flag} = Rd[\text{immediate}-1]$ $Rd = Rd \gg Rs, C\text{ flag} = Rd[Rs-1]$

ARM 嵌入式系统开发

MOV	把 32 位数送入寄存器	Rd = immediate Rd = Rn Rd = Rm
MUL	32 位乘法	Rd = (Rm * Rd)[31:0]
MVN	把一个 32 位数的“非”送到寄存器	Rd = NOT(Rm)
NEG	求反	Rd = 0 - Rm
ORR	32 位逻辑位“或”	Rd = Rd OR Rm
ROR	32 位循环右移	Rd = Rd 循环右移 Rs C flag = Rd[Rs-1]
SBC	32 位带进位减法	Rd = Rd - Rm - NOT(C flag)
SUB	32 位减法	Rd = Rn - immediate Rd = Rd - immediate Rd = Rn - Rm Sp = sp - (immediate << 2)
TST	32 位测试指令	Rn AND Rm set flags

这些指令与等价的 ARM 指令使用相同的格式。大多数的 Thumb 数据处理指令操作寄存器 r0~r7, 同时会更新 cpsr。但是下列指令是例外:

```
MOV    Rd, Rn
ADD    Rd, Rm
CMP    Rn, Rm
ADD    sp, #immediate
SUB    sp, #immediate
ADD    Rd, sp, #immediate
ADD    Rd, pc, #immediate
```

这些指令可以操作寄存器 r8~r14 和 pc。在使用 r8~r14 时, 除了 CMP 指令外, 其它指令不改变 cpsr 中的条件标志。CMP 指令总是更新 cpsr 的。

【例 4.3】 一个简单的 Thumb ADD 指令。

指令带有两个寄存器 r1 和 r2, 把它们相加后的结果放到寄存器 r0, 同时更新 cpsr 的值。

PRE

```
cpsr = nzcvtFT_SVC
r1 = 0x80000000
r2 = 0x10000000
```

```
ADD r0,    r1,    r2
```

POST

```
r0 = 0x90000000
cpsr = NzcvtFT_SVC
```

【例 4.4】与 ARM 方式不同, Thumb 的桶形移位操作(ASR, LSL, LSR 及 ROR)是单独的指令。这个例子显示了逻辑左移(LSL)指令, 把寄存器 r2 乘以 2。

PRE

```
r2 = 0x00000002
r4 = 0x000000001
```

```
LSL    r2,    r4
```

POST

```
r2 = 0x00000004
r4 = 0x00000001
```

完整的 Thumb 数据处理指令列表, 请参见附录 A。

4.5 单寄存器 load-store 指令

Thumb 指令集支持寄存器的装载和存储, 即 LDR 和 STR 指令。这些指令使用 2 种前变址寻址方式: 寄存器偏移和立即数偏移。

语法:

```
<LDR|STR>{<B|H>} Rd, [Rn, #immediate]
```

```
LDR{<H|SB|SH>} Rd, [Rn, Rm]
```

```
STR{<B|H>} Rd, [Rn, Rm]
```

```
LDR Rd, [pc, #immediate]
```

```
<LDR|STR> Rd, [sp, #immediate]
```

ARM 嵌入式系统开发

LDR	装载字到一个寄存器	$Rd \leftarrow \text{mem32}[\text{address}]$
STR	存储寄存器字	$Rd \rightarrow \text{mem32}[\text{address}]$
LDRB	装载字节到一个寄存器	$Rd \leftarrow \text{mem8}[\text{address}]$
STRB	存储寄存器中的字节	$Rd \rightarrow \text{mem8}[\text{address}]$
LDRH	装载半字到一个寄存器	$Rd \leftarrow \text{mem16}[\text{address}]$
STRH	存储寄存器中的半字	$Rd \rightarrow \text{mem16}[\text{address}]$
LDRSB	装载有符号字节到一个寄存器	$Rd \leftarrow \text{signExtend}(\text{mem8}[\text{address}])$
LDRSH	装载有符号半字到一个寄存器	$Rd \leftarrow \text{signExtend}(\text{mem16}[\text{address}])$

表 4.3 列出了不同的寻址方式。第 1 种寻址方式为寄存器偏移使用一个基址寄存器 R_n 加上寄存器偏移量 R_m ; 第 2 种寻址方式使用基址寄存器 R_n 加上一个 5 位的立即数或者一个依赖于数据尺寸的值。5 位的偏移在指令中的编码根据 8 位访问、16 位访问及 32 位访问, 分别乘以 1, 2 及 4。

表 4.3 寻址模式

类 型	语 法
load/store 寄存器	$[R_n, R_m]$
基址寄存器+偏移	$[R_n, \# \text{immediate}]$
相对寻址	$[pc sp, \# \text{immediate}]$

【例 4.5】 显示两个使用前变址寻址方式的 Thumb 指令, 它们执行前的条件是一样的。

PRE

```
Mem32[0x9000] = 0x00000001
```

```
Mem32[0x9004] = 0x00000002
```

```
Mem32[0x9008] = 0x00000003
```

```
r0 = 0x00000000
```

```
r1 = 0x00090000
```

```
r4 = 0x00000004
```

```
LDR    r0,    [r1,r4]    ;register
```

POST

```
r0 = 0x00000002
```

```
r1 = 0x00090000
```

```
r4 = 0x00000004
```

```
LDR    r0,    [r1, #0x4]    ;immediate
```

POST

```
r0 = 0x00000002
```

这 2 条指令执行同样的操作。惟一的区别是第 2 条 LDR 使用了一个固定偏移, 而第 1 条 LDR 的偏移依赖于寄存器 r4。

4.6 多寄存器 load-store 指令

Thumb 指令集的多寄存器 load-store 指令是 ARM 指令集的多寄存器 load-store 指令的简化形式。在 Thumb 指令集的多寄存器 load-store 指令只支持后增量 (IA, *Increment After*) 寻址方式。

语法:

$\langle \text{LDM} | \text{STM} \rangle \text{IA } \text{Rn!}, \{ \text{low register list}; \text{r0} \sim \text{r7} \}$

LDMIA	装载多个寄存器	$\{ \text{Rd} \} * N \leftarrow \text{mem32}[\text{Rn} + 4 * N], \text{Rn} = \text{Rn} + 4 * N$
STMIA	存储多个寄存器	$\{ \text{Rd} \} * N \rightarrow \text{mem32}[\text{Rn} + 4 * N], \text{Rn} = \text{Rn} + 4 * N$

这里 N 是寄存器列表中寄存器的数目。从表中可看到, 指令执行后总是更新基址寄存器, 基址寄存器和可以使用的寄存器列表仅限于 r0~r7。

【例 4.6】 保存 r1~r3 到内存地址 0x9000~0x900c, 并且更新基址寄存器 r4。
需要指出的是, 这里更新字符“!”不是可选的, 这与 ARM 指令集不同。

PRE

```
r1 = 0x00000001
```

```
r2 = 0x00000002
```

```
r3 = 0x00000003
```

```
r4 = 0x9000
```

```
STMIA    r4!,    {r1,r2,r3}
```

POST

```
mem32[0x9000]    = 0x00000001
```

```
mem32[0x9004]    = 0x00000002
mem32[0x9008]    = 0x00000003
r4 = 0x900c
```

4.7 堆栈指令

Thumb 的堆栈操作与等效的 ARM 指令是不同的,因为它们使用了更传统的 POP 和 PUSH 的概念。

语法:

```
POP {low_register_list{,pc}}
PUSH {low_register_list{,lr}}
```

POP	出 栈	$Rd * N \leftarrow mem32[sp + 4 * N], sp = sp + 4 * N$
PUSH	入 栈	$Rd * N \rightarrow mem32[sp + 4 * N], sp = sp - 4 * N$

注意:在指令中没有堆栈指针,这是因为在 Thumb 操作中,寄存器 r13 是固定作为堆栈指针用的,sp 是自动更新的。可操作的寄存器列表仅限于寄存器 r0~r7。

PUSH 指令可以操作的寄存器还包括链接寄存器 lr,同样 POP 指令可以操作 pc。这为子程序的进入和退出提供了支持,如例 4.7 所述。

堆栈指令仅支持递减式满堆栈操作。

【例 4.7】 使用 PUSH 和 POP 指令,子程序 ThumbRoutine 使用带链接的分支指令 (BL)来调用。

```
    ;调用子程序
    BL ThumbRoutine
    ;其它代码
ThumbRoutine
    PUSH    {r1, lr}    ;进入子程序
    MOV     r0, #2
    POP     {r1, pc}    ;从子程序返回
```

链接寄存器 lr 和 r1 被压入堆栈,在返回时,寄存器 r1 的值被原来的 r1 出栈恢复,pc 被原来入栈的 lr 的值覆盖。这就完成了从子程序返回。

4.8 软件中断指令

与 ARM 指令集下的软件中断指令相似,Thumb 软件中断指令(SWI)也产生一个软件中断异常。在 Thumb 状态下,如果有任何中断或者异常标志出现,那么处理器就会自动回到 ARM 状态去进行异常处理。

语法:

SWI immediate

SWI	软件中断	lr_svc=SWI之后的第一条指令的地址 spsr_svc=cpsr pc=vectors + 0x8 cpsr 模式=SVC cpsr I=1(屏蔽 IRQ 中断) cpsr T=0(ARM 状态)
-----	------	--

Thumb SWI 指令与等效的 ARM 指令有同样的作用和几乎完全相同的语法。区别是 Thumb SWI 数目限制在 0~255,并且不能条件执行。

【例 4.8】 显示 Thumb SWI 的执行。

注意: 在执行该指令后,处理器从 Thumb 状态切换到 ARM 状态。

PRE

```
cpsr = nzcVqift_USER
pc   = 0x00008000
lr   = 0x003ffffff,lr = r14
r0   = 0x12
```

0x00008000 SWI 0x45

POST

```
cpsr = nzcVqift_SVC
spsr = nzcVqift_USER
pc   = 0x00000008
lr   = 0x00008002
r0   = 0x12
```

4.9 总 结

本章主要介绍了 Thumb 指令集。所有的 Thumb 指令长度都是 16 位的,Thumb 代码可以提供比 ARM 代码高大约 30% 的代码密度。大多数 Thumb 代码都是由 C 或者 C++ 这样的高级语言编译而成的。

ATPCS 定义了 ARM 和 Thumb 代码如何相互调用,称为 *ARM-Thumb* 交互。交互使用分支切换指令 BX 和带链接的分支切换指令 BLX 来改变状态,并跳转到特定的例程。

在 Thumb 指令集中,只有分支指令可以条件执行,桶形移位操作(ASR, LSL, LSR 及 ROR)是单独的指令。

多寄存器 load-store 指令只支持后增量(IA)寻址方式。Thumb 指令集包括 POP 和 PUSH 指令,用以进行堆栈操作,这些指令只支持递减式满堆栈。

Thumb 指令不可以访问协处理器、cpsr 和 spsr。

第 5 章

高效的 C 编程

- C 编译器及其优化概述
- 基本的 C 数据类型
- C 循环结构
- 寄存器分配
- 函数调用
- 指针别名
- 结构体安排
- 位 域
- 边界不对齐数据和字节排列方式(大/小端)
- 除 法
- 浮点运算
- 内联函数和内嵌汇编
- 移植问题
- 总 结

本章将帮助读者在 ARM 处理器上编写高效的 C 代码。我们将通过许多小的例程来说明编译器如何把 C 代码转换成 ARM 汇编代码。当读者了解这种转换后,就可以区分出执行速度快和慢的 C 代码。这些例子都是基于普通 C 语言的,但这些技术也同样适用于 C++ 语言。

本章首先介绍 C 编译器及其优化,帮助读者理解 C 编译器在优化代码时所碰到的一些问题。理解这些问题,将有助于编写出在提高执行速度和减小代码尺寸方面更高效的 C 源代码。后面的小节是按主题来组织的。

5.2 和 5.3 节以一个数据包校验和的简单程序为例,分析、说明了如何优化一个基本的 C 循环。5.4 和 5.5 节介绍了如何优化一个完整的 C 函数体,包括在一个函数内编译器如何分配寄存器,如何减少函数调用时的开销。

5.6~5.9 节分析了有关存储器操作的问题,包括指针操作、数据打包和高效的存储器访问。5.10~5.12 节描述了 ARM 指令通常不直接支持的一些基本操作,也可以使用内联函数和内嵌汇编来增加自己的基本操作。

最后总结了把 C 代码从其它的体系结构移植到 ARM 结构时会碰到的一些问题。

5.1 C 编译器及其优化概述

本章假定读者熟悉 C 语言,也有一些汇编语言编程方面的知识。后者虽然不是必需的,但是对于理解后面例子编译后的汇编输出结果会有帮助。ARM 汇编语法的详细内容可参见第 3 章或附录 A。

众所周知,优化代码需要花费时间,而且会降低源代码的可读性。所以通常只对经常被调用且对性能影响较大的函数进行优化。为了找到这些函数,推荐使用大多数 ARM 编译和调试器都带的性能分析工具。另外,用源代码注释来评注那些不容易理解的优化代码,可以提高代码的可维护性。

C 编译器必须逐字逐句地把 C 程序转换成汇编程序,这样编译器就不会漏掉所有可能的输入。实际上,许多输入组合是不可能的或不会出现的。首先来看一个例子,函数 `memclr()` 用来清除从地址 `data` 开始的 `n` 字节的存储单元内容。从这个例子中可以看到编译器会碰到的问题。

```
void memclr(char *data, int N)
{
    for ( ; N>0 ; N-- )
    {
        *data = 0;
```

```

    data++;
}
}

```

首先,编译器无论多高级,也不可能知道 N 的输入值是否可以是 0。因此,在第一个循环开始之前,编译器需要对这个问题进行明确的检查。

其次,编译器也不知道数组指针 $data$ 是否是 4 字节边界对齐的。如果是 4 字节对齐的,那么编译器就可以使用 `int` 而不是 `char` 类型的指针,这样一次就可以清除 4 字节的存储单元。而且,编译器也不知道 N 是否是 4 的整数倍,如果 N 是 4 的整数倍,那么编译器可以重复循环体中的内容 4 次或者利用 `int` 类型的指针一次存储 4 字节。

然而,编译器必须是保守的,只能假定 N 的所有可能的值和 $data$ 所有可能的边界值。这些问题将在 5.3 节中详细讨论。

总之,要编写高效的 C 代码,必须了解哪些地方 C 编译器是保守的,编译器涉及到的处理器结构的限制,以及一些特殊的 C 编译器限制。

本章大部分内容都围绕着上述前两点,并适用于所有的 ARM C 编译器。第 3 点就要依赖于编译器供应商和编译器的修订版本了,读者须参考编译器的有关文档,或者亲自对编译器做各种测试。

为了保证例子的一致性,已经用下面的 C 编译器测试过所有程序:

- ARM Developer Suite version 1.1(ADS 1.1)的 `armcc` 可直接从 ARM 购买这个版本或后续版本的使用许可。
- `Arm-elf-gcc version 2.95.2` 是 GNU C 编译器的 ARM 版本 `gcc`,是免费使用的。

在本书中,使用 ADS1.1 下的 `armcc` 来生成例子中的汇编输出结果。下面的脚本显示了如何对 C 文件 `test.c` 使用 `armcc`。可以使用这种方法来重新生成程序的编译结果。

```

armcc -Otime -c -o test.o test.c
fromelf -text/c test.o > test.txt

```

`armcc` 默认是全部优化功能有效(`-O2` 命令行选项)。`-Otime` 选项表示执行速度优化高于代码空间的优化,这主要是影响编译器针对 `for` 和 `while` 循环的处理。如果使用 `gcc` 编译器,那么下面的脚本可以生成类似的汇编输出:

```

arm-elf-gcc -O2 -fomit-frame-pointer -c -o test.o test.c
arm-elf-objdump -d test.o > test.txt

```

GNU 编译器在默认状态下所有优化都是关掉的。`-fomit-frame-pointer` 选项阻止 GNU 编译器保留结构指针寄存器。结构指针可以帮助调试窗口显示存储在堆栈的局部变量。但是,如果保留,那么效率将会降低,所以对性能有要求的代码就不要使用结构指针。

5.2 基本的 C 数据类型

首先来研究一下 ARM 编译器如何处理基本的 C 数据类型,将会发现其中一些数据类型用作局部变量时,执行效率要比其它类型的高。同时,在一定的寻址模式下,装载/存储不同类型的数据,效率是不一样的。

ARM 处理器内部是 32 位寄存器和 32 位的数据处理操作。其体系结构是 RISC load/store 结构。换句话说,数据在使用前必须先将其从内存装载到内部寄存器,任何算术或者逻辑指令都不能直接在存储器里进行数据操作。

早期版本的 ARM 结构(ARMv1~v3)为装载和存储无符号 8 位和无符号/有符号 32 位数据提供了硬件支持。这些体系结构都是用于比 ARM7TDMI 更早的处理器。表 5.1 指出了 ARM 体系结构支持的 load/store 指令。

表 5.1 ARM 体系结构的 load 和 store 指令

体系结构	指 令	执行的功能
Pre-ARMv4	LDRB	装载一个无符号的 8 位数据
	STRB	存储一个有/无符号的 8 位数据
	LDR	装载一个有/无符号的 32 位数据
	STR	存储一个有/无符号的 32 位数据
ARMv4	LDRSB	装载一个有符号的 8 位数据
	LDRH	装载一个无符号的 16 位数据
	LDRSH	装载一个有符号的 16 位数据
	STRH	存储一个有/无符号的 16 位数据
ARMv5	LDRD	装载一个有/无符号的 64 位数据
	STRD	存储一个有/无符号的 64 位数据

在表 5.1 中,8 位或者 16 位数据在装载/存储 ARM 寄存器之前,先要扩展成 32 位。无符号数把 0 作为扩展位,有符号数则按照符号位扩展。这就意味着装载 int 类型(32 位)的数据无须花费多余的指令时间来进行位扩展。同样在存储时,8 位或者 16 位的数据必须放到寄存器的低 8 位或低 16 位。而一个 int 或更小类型的传送,存储时就不需要花费额外的指令时间了。

ARMv4 及其以后的体系结构可以使用新增的指令来直接装载和存储一个带符号 8 位

和 16 位数据。由于这些指令都是后来增加的,它们并不支持早于 ARMv4 指令集的许多寻址方式(不同寻址方式详见 3.3 节)。在后面 5.2.1 小节中的例子 checksum_v3 将会看到这种影响。

ARMv5 增加了支持 64 位数据的 load 和 store 指令,ARM9E 及其以后的核都支持这些指令。

比 ARMv4 早的 ARM 处理器并不能很好地处理有符号的 8 位或者任何 16 位数据。因此,在 ARM C 编译器中定义的 char 类型是 8 位无符号的,而不像其它编译器默认是 8 位有符号的。

在表 5.2 中,说明了 armcc 和 gcc 编译器对 ARM 文件所使用的数据类型映射。当把代码从其它体系结构的处理器移植到 ARM 处理器时,对于 char 数据类型要特别注意,因为它可能会引入一些问题。比如经常使用一个 char 类型的数据 i 作为循环计数器,循环的持续条件是 $i \geq 0$ 。由于 i 对 ARM 编译器来说是一个无符号的数,这个循环将永远不会结束。幸运的是,armcc 在这种情况下会给出一个警告:unsigned comparison with 0。另外,编译器也提供了补充选项,可以使 char 类型变成带符号的。例如,在 gcc 中命令行选项 -fsigned-char 就可以将 char 转换成有符号类型。在 armcc 中命令行选项 -zc 也有同样的效果。

与本书的其它部分一样,这里都是假定使用 ARMv4 体系结构及其以上的处理器,包括 ARM7TDMI 及其以后所有的处理器。

表 5.2 C 编译器数据类型映射

C 数据类型	表示的意义
char	无符号 8 位字节数据
short	有符号 16 位半字数据
int	有符号 32 位字数据
long	有符号 32 位字数据
long long	有符号 64 位双字数据

5.2.1 局部变量类型

基于 ARMv4 体系结构的处理器能高效地装载和存储 8 位、16 位和 32 位数据。但是,大多数的 ARM 数据处理操作都是 32 位的。基于这个原因,局部变量应尽可能使用 32 位的数据类型 int 或者 long。即使在处理 8 位或者 16 位的数值时,也应避免使用 char 和 short 数据类型作为局部变量。惟一的例外情况是,需要使用 char 或者 short 类型的数据溢

出归零特性时,如模运算 $255+1=0$,就要使用 char 类型。

为了说明局部变量类型的影响,先来看一个简单的例子:checksum(校验和)函数的功能是计算一个数据包内的数据总和。这个例子具有很普遍的意义,因为大多数的通信协议(例如 TCP/IP)都有校验和或者循环冗余校验(CRC)程序来检查一个数据包里是否有错误。

下面的程序用来计算一个包含 64 个字的数据包的校验和。其说明了为什么对局部变量应该避免使用 char 类型。

```
int checksum_v1 (int * data)
{
    char i;
    int sum = 0;

    for ( i = 0 ; i < 64 ; i++)
    {
        sum += data[i] ;
    }

    return sum;
}
```

初看一下,似乎声明 i 为 char 类型是没有什么问题的,甚至可能会觉得一个 char 类型的数据比 int 类型的数据占用更小的寄存器空间或者更小的 ARM 堆栈空间。其实对 ARM 来说,这两个设想都是错误的。所有的 ARM 寄存器都是 32 位的,所有的堆栈入口也至少是 32 位的。而且,为了正确执行 $i++$,编译器必须解决 $i=255$ 时的情况,因为对于 char 数据类型的 i 来说,255 加 1 产生的结果是 0。

这个函数经过编译后的输出结果如下,这里增加了标号和注释,以使汇编语言更加清晰。

```
checksum_v1
    MOV     r2,r0                ;r2 = data
    MOV     r0,#0                ;sum = 0
    MOV     r1,#0                ;i = 0
checksum_v1_loop
    LDR     r3,[r2,r1,LSL #2]    ;r3 = data [i]
    ADD     r1,r1,#1            ;r1 = i + 1
    AND     r1,r1,#0xff          ;i = (char) r1
    CMP     r1,#0x40             ;compare i,64
    ADD     r0,r3,r0             ;sum += r3
    BCC     checksum_v1_loop    ;if ( i < 64 ) loop
    MOV     pc,r14               ;return sum
```


现在把上面的程序段与把 `i` 声明为 `unsigned int` 类型时比较一下:

```
checksum_v2
    MOV    r2,r0                ;r2 = data
    MOV    r0,#0                ;sum = 0
    MOV    r1,#0                ;i = 0
checksum_v2_loop
    LDR    r3,[r2,r1,LSL #2]    ;r3 = data[i]
    ADD    r1,r1,#1             ;r1++
    CMP    r1,#0x40             ;compare i,64
    ADD    r0,r3,r0             ;sum += r3
    BCC    checksum_v2_loop     ;if ( i < 64 ) goto loop
    MOV    pc,r14               ;return sum
```

第一种情况,在 `i` 和 64 比较前,编译器增加了额外的 `AND` 指令来保证 `i` 的范围为 0~255。在第二种情况下,这条指令就可以省略了。

接下来,假设数据包中的数据是 16 位的,需要计算一个 16 位的校验和功能。试写出下面的 C 代码:

```
short checksum_v3 ( short * data)
{
    unsigned int i;
    short sum = 0;

    for ( i=0; i<64; i++)
    {
        sum = (short) ( sum + data[i] );
    }
    return sum;
}
```

读者可能会疑惑,为什么循环体内的代码不是 `sum += data[i]`;如果选择了 `armcc` 编译器中隐式数据宽度窄化警告(implicit narrowing cast warning)选项 `-W+n`,那么编译上述代码时就会出现一个警告。表达式 `sum + data[i]` 是整数类型的,所以只能进行数据宽度窄化(隐式或者显式),再赋给一个 `short` 类型的数据。见下面的汇编输出结果,注意编译器必须增加指令来实现数据宽度窄化:

```
checksum_v3
    MOV    r2,r0                ;r2 = data
```

```

MOV    r0, #0                ;sum = 0
MOV    r1, #0                ;i = 0
checksum_v3_loop
    ADD    r3, r2, r1, LSL #1    ;r3 = &data[i]
    LDRH   r3, [r3, #0]         ;r3 = data[i]
    ADD    r1, r1, #1          ;i++
    CMP    r1, #0x40           ;compare i, 64
    ADD    r0, r3, r0          ;r0 = sum + r3
    MOV    r0, r0, LSL #16
    MOV    r0, r0, ASR #16      ;sum = (short) r0
    BCC    checksum_v2_loop     ;if (i < 64) goto loop
    MOV    pc, r14             ;return sum

```

这个循环要比前面的例子 checksum_v2 的循环多了 3 条指令！对于额外的指令有以下两种解释：

- LDRH 指令不能像 checksum_v2 中的 LDR 指令一样支持移位地址偏移(shifted address offset)，因此在循环体中第一条 ADD 指令用来计算数组下标 i 的地址。LDRH 指令只能从一个没有偏移量的地址装入数据。由于 LDRH 指令是 ARM 指令集中后来增加的，所以相对于 LDR 指令，它的寻址方式较少(见表 5.1)。
- 使 total + array[i] 变为 short 类型需要 2 条 MOV 指令。编译器先左移 16 位，然后右移 16 位，以实现一个 16 位符号扩展。右移是符号位扩展移位，它复制了符号位来填充高 16 位。

要避免第二个问题，可以使用 int 类型的变量来计算、保持校验和，只是在函数退出时，将和值转换为 short 类型。

然而，第一个问题是新出现的。要解决这个问题，可以通过递增指针 data，而不是用数组 data[i] 的下标来访问数组。这是忽视数组类型长度或元素尺寸的有效方法。ARM 所有的 load 和 store 指令都支持后增量(postincrement)寻址方式。

【例 5.1】 checksum_v4 代码解决了本节讨论的所有问题。局部变量使用了 int 类型，避免了不必要的转换，并且使用了指针 data 代替原来使用数组的下标。

```

short checksum_v4 (short * data )
{
    unsigned int i ;
    int sum = 0 ;

    for ( i = 0 ; i < 64 ; i++)
    {

```

```

        sum += * ( data ++ );
    }
    return (short) sum;
}

```

操作 `* (data++)` 只需一条 ARM 指令装载数据并增加指针 `data` 的值。当然,也可以根据各自的喜爱,写成 `sum += data; data++;` 或者 `* data++`。编译器会产生下面的输出结果。相对于 `checksum_v3`,在内部循环体中 3 条指令被删除了,这样对于每个循环来说节约了 3 个周期。

```

checksum_v4
    MOV     r2, #0                ;sum = 0
    MOV     r1, #0                ;i = 0
checksum_v4_loop
    LDRSH   r3, [r0], #2          ;r3 = * ( data ++ )
    ADD     r1, r1, #1            ;i ++
    CMP     r1, #0x40             ;compare i, 64
    ADD     r2, r3, r2            ;sum += r3
    BCC     checksum_v4_loop      ;if ( sum < 64 ) goto loop
    MOV     r0, r2, LSL #16
    MOV     r0, r0, ASR #16       ;r0 = (short) sum
    MOV     pc, r14              ;return r0

```

在函数返回时,编译器仍然需要把指令数据转换成 16 位的。如果返回值是 `int` 类型,那么这一步就不需要了。5.2.2 小节将进一步讨论返回值是 `int` 类型的情况。

101

5.2.2 函数参数类型

在 5.2.1 小节中已经看到,把局部变量从 `char` 或者 `short` 类型转换成 `int` 类型,可以改善性能并减小代码尺寸。其实,这种转换对函数参数也有着同样的效果。看下面的例子,将 2 个 16 位的值相加,其中第 2 个数先减半,然后返回一个 16 位的和:

```

short add_v1 ( short a, short b )
{
    return a + ( b >> 1 );
}

```

虽然这个函数只是一个简单的算术运算,但是对于观察编译器所碰到的问题是一个很好的例子。输入值 `a`, `b` 和返回值都存放在 32 位的 ARM 寄存器中。编译器是否应该考虑

ARM 嵌入式系统开发

这些 32 位数值在 short 类型的范围($-32768 \sim +32767$)之间呢? 或者编译器是否应该通过对低 16 位数据进行符号位扩展, 充填 32 位寄存器, 强制把数值限制在上述范围之内呢? 编译器必须对函数调用者和被调用者作出一致的决策。不是调用者, 就是被调用者, 必须把数据转换为 short 类型。

如果参数可以不缩小到所定义的数据类型范围, 那么称这种函数参数传递是宽的(wide); 反之, 则称为窄的(narrow)。观察 add_v1 的汇编输出结果, 就可以知道编译器所采用的是哪种形式。如果编译器传递参数是宽的, 那么被调用者就必须把参数缩小到正确的范围; 如果编译器传递参数是窄的, 那么调用者就必须缩小参数范围。如果编译器返回值是宽的, 那么调用者就必须把返回值缩小到正确的范围; 如果编译器返回值是窄的, 那么被调用者就必须在返回前缩小返回值的范围。

对于 ADS 的 armcc 来说, 函数参数传递和返回值都是窄的。换句话说, 调用者要处理调用参数, 而被调用者要处理返回值。编译器采用函数的 ANSI 原型来决定函数参数的数据类型。

函数 add_v1 在 armcc 下的输出结果显示, 编译器把返回值的类型转换为 short, 而没有处理输入参数。它认为调用者已经保证在 r0 和 r1 中的 32 位的数值是在 short 类型的范围之内。这就说明参数传递和返回值都是窄的。

```
add_v1
    ADD     r0,r0,r1,ASR #1      ;r0 = (int)a + ((int) b >> 1)
    MOV     r0,r0,LSL #16
    MOV     r0,r0,ASR #16      ;r0 = ( short ) r0
    MOV     pc,r14              ;return r0
```

gcc 编译器更为谨慎, 对参数值的范围不作任何假设。这个版本的编译器, 在调用者和被调用者中都将输入参数缩小到 short 类型的数据范围, 同时也都将返回值转换为 short 类型。下面是 add_v1 由 gcc 编译后的代码:

```
add_v1_gcc
    MOV     r0,r0,LSL #16
    MOV     r1,r1,LSL #16
    MOV     r1, r1,ASR #17      ;r1 = ( int ) b >> 1
    ADD     r1,r1,r0,ASR #16    ;r1 += ( int ) a
    MOV     r1,r1,LSL #16
    MOV     r0, r1,ASR #16      ;r0 = (short) r1
    MOV     pc,lr              ;return r0
```

尽管宽和窄的函数调用规则各有其优点, 但 char 或者 short 类型的函数参数和返回值都会产生额外的开销, 导致性能的下降, 并增加了代码尺寸。所以, 即使是传输一个 8 位的

数据,函数参数和返回值使用 int 类型也会更有效。

5.2.3 有符号数与无符号数

前面说明了对于局部变量和函数参数,使用 int 类型比使用 char 或者 short 类型更好。本小节将对有符号整数(signed int)和无符号整数(unsigned int)类型的执行效率进行分析比较。

如果程序中只有加法、减法及乘法,那么有符号和无符号数的执行效率没有任何差别。但是,如果有了除法,就不一样了。看下面一个关于求 2 个整数平均值的简单例子:

```
int average_v1 ( int a,int b )
{
    return ( a + b ) / 2 ;
}
```

编译后:

```
average_v1
    ADD        r0,r0,r1                ;r0 = a+b
    ADD        r0,r0,r0,LSR #31        ;if (r0<0) r0++
    MOV        r0,r0,ASR #1            ;r0 = r0 >> 1
    MOV        pc,r14                  ;return r0
```

注意:在汇编代码中,如果和(r0)是负数,则在右移前编译器先对 r0 进行了加 1 操作。换句话说,就是把 $x/2$ 替换成:

```
( x<0 ) ? (( x+1 ) >> 1 ) : ( x >> 1 )
```

这一步是必须做的,因为 x 是有符号数。在 ARM C 中,如果 x 是负数,那么除 2 操作就不是一个右移操作。例如: $-3 \gg 1 = -2$,然而 $-3/2 = -1$ 。对于一个负整数,除法的结果是向 0 的方向舍入,而算术右移的结果是向 -8 的方向舍入。

对于除法来说,使用无符号数效率会更高。编译器可以将 2 的倍数的无符号除法直接用右移来替代。对于一般的除法运算,C 库文件中的除法程序对于无符号数运算执行速度也更快。关于避免除法运算的讨论,参见 5.10 节。

摘 要 C 数据类型的有效用法

- 对于存放在寄存器中的局部变量,除了 8 位或 16 位的算术模运算外,尽量不要使用 char 和 short 类型,而要使用有符号或者无符号 int 类型。除法运算时使用无符号数执行速度更快。
- 对于存放在主存储器中的数组和全局变量,在满足数据大小的前提下,应尽可能使用小尺寸的数据类型,这样做可以节省存储空间。ARMv4 体系结构可以有效地装载和

ARM 嵌入式系统开发

存储所有宽度的数据,并可以使用递增数组指针来有效地访问数组。对于 short 类型数组,要避免使用数组基地址的偏移量,因为 LDRH 指令不支持偏移寻址。

- 通过读取数组或者全局变量并赋给不同类型的局部变量时,或者把局部变量写入不同类型的数组或者全局变量时,要进行显式(explicit)数据类型转换。这种转换使编译器可以明确、快速地处理,把存储器中数据宽度比较窄的数据类型扩展,并赋给寄存器中较宽的类型。通过打开编译器的隐式数据宽度窄化警告(implicit narrowing cast warning)选项,可以观察到隐式数据类型转换。
- 由于隐式或者显式的数据类型转换通常会有额外的指令周期开销,所以在表达式中应尽量避免使用。load 和 store 指令一般不会产生额外的转换开销,因为 load 和 store 指令是自动完成数据类型转换的。
- 对于函数参数和返回值应尽量避免使用 char 和 short 类型。即使参数范围比较小,也应该使用 int 类型,以防止编译器做不必要的类型转换。

5.3 C 循环结构

循环体是程序设计与优化的重点考虑对象。本节将着重于研究在 ARM 上处理 for 和 while 循环最有效的方法。首先研究循环次数固定大小的循环体,然后转移到可变次数的,最后将分析循环体展开。

5.3.1 固定次数的循环

什么是 ARM 上编写 for 循环效率最高的方法?再回到前面的 checksum 例子,研究一下循环结构。

下面是在 5.2 节中讨论的 64 个字数据包校验和程序的最后版本,说明了编译器如何使用增量计数 `i++` 来处理一个循环。

```
int checksum_v5 (int * data)
{
    unsigned int i;
    int sum = 0;

    for ( i = 0 ; i < 64 ; i ++ )
    {
        sum += *(data ++ );
    }
}
```

```

    }
    return sum;
}

```

编译后生成：

```

checksum_v5
    MOV     r2,r0                ;r2 = data
    MOV     r0,#0                ;sum = 0
    MOV     r1,#0                ;i = 0
checksum_v5_loop
    LDR     r3,[r2], #4          ;r3 = *(data++)
    ADD     r1,r1,#1             ;i++
    CMP     r1,#0x40             ;compare i,64
    ADD     r0,r3,r0             ;sum += r3
    BCC     checksum_v5_loop     ;if ( i < 64) goto loop
    MOV     pc,r14               ;return sum

```

这里用了 3 条指令来实现 for 循环结构：

- 一条 ADD 指令，增加 i 的值；
- 一条 CMP 比较指令，检查 i 是否小于 64；
- 一条 BCC 条件分支指令，如果 $i < 64$ ，则继续循环。

这种循环执行效率不高。在 ARM 上，一个循环其实只要 2 条指令就够了：

- 一条减法指令，进行循环减计数，同时设置结果的条件标志；
- 一条条件分支指令。

这里的关键是，循环的终止条件应为减计数到零(count down to zero)，而不是计数增加到某个特定的限制值。由于减计数结果已存储在条件标志里，与零比较的指令就可以省略了。由于不再使用 i 作为数组的下标索引，采用减计数就没有任何问题。

【例 5.2】 表明采用减计数循环(decrementing loop)要比增计数循环(incrementing loop)更好。

```

int checksum_v6 (int *data)
{
    unsigned int i;
    int sum = 0;

    for ( i = 64; i != 0 ; i--)

```

```

    {
        sum += *(data++);
    }

    return sum;
}

```

编译后输出:

```

checksum_v6
    MOV     r2,r0                ;r2 = data
    MOV     r0,#0                ;sum = 0
    MOV     r1,#0x40             ;i = 64
checksum_v6_loop
    LDR     r3,[r2],#4           ;r3 = *(data++)
    SUBS    r1,r1,#1             ;i-- and set flags
    ADD     r0,r3,r0             ;sum += r3
    BNE     checksum_v6_loop     ;if ( i != 0 ) goto loop
    MOV     pc,r14               ;return sum

```

这里, SUBS 和 BNE 指令实现了循环。这个校验和例程中, 每次循环只用了 4 条指令, 比 checksum_v1 中的 6 条和 checksum_v3 中的 8 条减少了许多。

对无符号的循环计数值 i 来说, 循环继续的条件既可以是 $i \neq 0$, 也可以是 $i > 0$ 。由于 i 不可能是负数, 所以这两个条件是等价的。而对一个有符号的循环计数值来说, 用条件 $i > 0$ 来作为继续循环的条件是一件冒险的事情。读者可能会期望编译器生成下面的 2 条指令来实现循环:

```

SUBS      r1,r1,#1              ;compare i with 1, i = i-1
BGT       loop                  ;if ( i+1>1 ) goto loop

```

实际上, 编译器生成的是下面的代码:

```

SUB       r1,r1,#1              ;i --
CMP       r1,#0                 ;compare i with 0,
BGT       loop                  ;if ( i>0 ) goto loop

```

不是编译器的无能, 当 $i = -0x80000000$ 时, 它必须小心, 因为上面 2 段汇编代码在这种情况下会产生不同的结果。对于第 1 段汇编代码, SUBS 指令将 i 与 1 比较, 然后减 1。由于 $-0x80000000 < 1$, 循环终止。而对于第 2 段汇编代码, 先将 i 减 1, 然后与 0 比较。模运算意味着现在 i 的大小是 $+0x7fffffff$, 这样循环就会继续下去。

当然, 实际上 i 很少会是 $-0x80000000$ 。编译器一般不可能碰到这种情况, 特别是当循

环是以一个可变的计数值开始时(参见 5.3.2 小节)。

总而言之,无论对于有符号还是无符号的循环计数值,都应使用 $i \neq 0$ 作为循环的结束条件。对有符号数 i ,这比使用条件 $i > 0$ 少了一条指令。

5.3.2 不定次数的循环

现在假定 checksum 程序要处理任意长度的数据包,这样就须传递一个变量 N 作为数据包的长度。从 5.3.1 小节的经验可知,可使用减计数直至 $N=0$,也不需要额外的循环计数值 i 。

例子 checksum_v7 显示编译器如何处理不定次数 N 的循环。

```
int checksum_v7 (int *data, unsigned int N)
{
    int sum = 0;

    for (; N != 0 ; N--)
    {
        sum += *(data++);
    }
    return sum;
}
```

编译后生成:

```
checksum_v7
    MOV     r2, #0                ;r2 = data
    CMP     r1, #0                ;compare N,0
    BEQ     checksum_v7_end       ;if ( N == 0 ) goto end
checksum_v7_loop
    LDR     r3, [r0], #4          ;r3 = *(data++)
    SUBS    r1, r1, #1            ;N-- and set flags
    ADD     r2, r3, r2            ;sum += += r3
    BNE     checksum_v7_loop      ;if ( N != 0 ) goto loop
checksum_v7_end
    MOV     r0, r2                ;r0 = sum
    MOV     pc, r14               ;return r0
```

注意:在函数的入口处,编译器先检查 N 是否为 0。由于数组通常不会为空,所以一般来说检查是不必要的。下面的例子说明使用 do-while 循环会比 for 循环呈现出更好的性能和代码密度。

【例 5.3】 显示如何使用 do-while 循环去除 for 循环中出现的对 N 是否为 0 的判断。

```

int checksum_v8 (int * data,unsigned int N)
{
    int sum = 0;

    do
    {
        sum += *(data++);
    } while ( --N != 0 );
    return sum;
}

```

编译后变成:

```

checksum_v8
    MOV     r2,#0                                ;sum = 0
Checksum_v8_loop
    LDR     r3,[r0],#4                            ;r3 = *(data++)
    SUBS    r1,r1,#1                            ;N-- and set flags
    ADD     r2,r3,r2                            ;sum += r3
    BNE     checksum_v8_loop                    ;if ( N != 0) goto loop
    MOV     r0,r2                                ;r0 = sum
    MOV     pc,r14                              ;return r0

```

与 checksum_v7 编译生成的代码比较,可发现减少了 2 个周期。

5.3.3 循环展开

在 5.3.1 小节中可以发现,每次循环需要在循环体外加 2 条指令:一条减法指令来减少循环计数值和一条条件分支指令。通常把这些指令称为循环开销(loop overhead)。在 ARM7 或者 ARM9 处理器上,减法指令需要 1 个周期,条件分支指令需要 3 个周期,这样每个循环就需要 4 个周期的开销。

可通过展开(unrolling)循环体——重复循环主体多次,并按同样的比例减少循环次数,来降低循环开销。例如,把数据包校验和程序展开 4 次。

【例 5.4】 把数据包校验和程序展开 4 次。

假定数据包中的数据个数 N 是 4 的倍数。

```

int checksum_v9 (int * data,unsigned int N)
{

```

```

int sum = 0;

do
{
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    N -= 4;
} while ( N != 0 );
return sum;
}

```

编译后生成:

```

checksum_v9
    MOV    r2, #0                      ;sum = 0
checksum_v9_loop
    LDR     r3,[r0], #4                 ;r3 = *(data++)
    SUBS    r1,r1,#4                   ;N -= 4 and set flags
    ADD     r2, r3,r2                  ;sum += r3
    LDR     r3,[r0], #4                 ;r3 = *(data++)
    ADD     r2, r3,r2                  ;sum += r3
    LDR     r3,[r0], #4                 ;r3 = *(data++)
    ADD     r2, r3,r2                  ;sum += r3
    LDR     r3,[r0], #4                 ;r3 = *(data++)
    ADD     r2, r3,r2                  ;sum += r3
    BNE     checksum_v9_loop           ;if ( N != 0) goto loop
    MOV     r0,r2                      ;r0 = sum
    MOV     pc,r14                     ;return r0

```

这里把循环开销从 $4N$ 个周期减少到 $4N/4=N$ 个周期。在 ARM7TDMI 上,把原来一次累加需要的 8 个周期减少到 $20/4=5$ 个周期,几乎加速了 2 倍! 对于 ARM9TDMI,由于它有更快的 load 指令,效果会更好。

在展开一个循环时,读者一定会问到 2 个问题:

- 到底应该展开循环几次?
- 如果循环的次数不是循环展开数的倍数该怎么办? 例如,在例 checksum_v9 中,如果 N 不是 4 的倍数怎么办?

首先,对于第 1 个问题,只有当循环展开对提高应用程序的整体性能非常重要时,才进行循环展开;否则得益不大反而会增加代码尺寸,甚至会因为替换掉了 cache 中更重要的代码而降低了总体性能。

假定循环对整个程序的性能是十分重要的,比如,占了整个程序执行的 30%;假设展开循环至 0.5 KB 的代码量(128 条指令)。这样,相对于循环主体的 128 个周期,循环的一般开销最多只有 4 个周期,占了 $3/128$,约 3%,而循环占整个程序的 30%。这样,循环的一般开销只占了整个程序执行的 1%。进一步展开代码对性能只能造成微乎其微的好处,却对 cache 内容造成了很大的影响。通常,如果对性能改善小于 1%,那么循环就不值得进一步展开了。

对于第 2 个问题,应设法使循环的次数是循环展开数的倍数。如果难以实现,那么就要增加额外的代码来处理数组的剩余元素。这将增加少许代码量,但可以保持较好的性能。

【例 5.5】 采用 4 次展开的循环,可对任意大小的数据包进行校验和计算。

```
int checksum_v10 (int *data,unsigned int N)
{
    unsigned int i;
    int sum = 0;

    for ( i = N/4 ; i != 0 ; i--)
    {
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
    }
    for ( i = N&3 ; i != 0 ; i--)
    {
        sum += *(data++);
    }
    return sum;
}
```

第 2 个 for 循环解决了当 N 不是 4 的倍数时的数组剩余元素问题。

注意: $N/4$ 和 $N\&3$ 都可能是零,所以就不能使用 do-while 结构的循环。

小结 高效地编写循环体

- 使用减计数到零的循环结构,这样编译器就不需要分配一个寄存器来保存循环终止值,而且与 0 比较的指令也可以省略;

- 使用无符号的循环计数值,循环继续的条件为 $i!=0$ 而不是 $i>0$,这样可以保证循环开销只有两条指令;
- 如果事先知道循环体至少会执行一次,那么使用 do-while 循环要比 for 循环好,这样可以使编译器省去检查循环计数值是否为零的步骤;
- 展开重要的循环体可降低循环开销,但不要过度展开,如果循环的开销对整个程序来说占的比例很小,那么循环展开反而会增加代码量并降低 cache 的性能;
- 尽量使数组的大小是 4 或 8 的倍数,这样就可以容易地以 2,4,8 次等多种选择展开循环,而不需要担心剩余数组元素的问题。

5.4 寄存器分配

编译器会试图对 C 函数中的每一个局部变量分配一个寄存器。如果几个局部变量不会交迭使用,那么编译器会对它们分配同一个寄存器。当局部变量多于可用的寄存器时,编译器会把多余的变量存储到堆栈。由于这些变量被写入了存储器,所以被称为溢出(spilled)或者替换(swapped out)变量,就像虚拟存储器的内容被替换到硬盘一样。与分配在寄存器中的变量相比,对溢出变量的访问要慢得多。

为了高效地执行一个函数,应该做到:

- 使溢出变量的数量最少;
- 确保最重要的和经常用到的变量被分配在寄存器。

首先,来了解一下 ARM C 编译器能够分配给局部变量的寄存器数目。表 5.3 显示了当 C 编译器采用 ARM-Thumb 过程调用标准(ATPCS)时,内部寄存器的编号、名字和分配方法。

表 5.3 C 编译器寄存器用法

寄存器编号	可选寄存器名字	ATPCS 寄存器用法
r0	a1	参数寄存器,在调用函数时,用来存放前 4 个函数参数和返回值。在函数内,如果把这些寄存器作为临时过渡寄存器来使用,则会破坏它们的值
r1	a2	
r2	a3	
r3	a4	
r4	v1	通用变量寄存器。调用函数必须保存被调用函数存放在这些寄存器中的变量值
r5	v2	
r6	v3	
r7	v4	
r8	v5	

续表 5.3

寄存器编号	可选寄存器名字	ATPCS 寄存器用法
r9	v6 sb	通用变量寄存器。在与读/写位置无关(RWPI)的编译情况下,r9 中保存静态基本地址,这个地址是读/写数据的地址;否则必须保存这个寄存器中被调用函数的变量值
r10	v7 sl	通用变量寄存器。在使用堆栈边界检查的编译情况下,r10 保存堆栈边界的地址;否则必须保存这个寄存器中被调用函数的变量值
r11	v8 fp	通用变量寄存器。除了在使用结构指针的编译情况下,必须保存这个寄存器中调用函数的变量值外,只有老版本的 armcc 编译器使用一个结构指针
r12	ip	通用临时过渡寄存器(草稿板)寄存器,函数调用时会破坏其中的值
r13	sp	堆栈指针,指向下降式满堆栈的堆栈
r14	lr	连接寄存器。在函数调用时,这个寄存器保存返回地址
r15	pc	程序计数器

假如编译器不使用软件堆栈检查或结构指针(frame pointer),那么 C 编译器就可以使用寄存器 r0~r12 和 r14 来存放变量。如果要用到这些寄存器,那么就必須用堆栈来保存 r4~r11 和 r14 中的值。

理论上,C 编译器可以分配 14 个变量到寄存器而不会溢出。实际上,一些编译器对某些寄存器有特定的用途,例如用 r12 作为临时过渡寄存器(草稿板)使用,编译器就不再分配任何变量给它了。另外,对于复杂的表达式,也需要过渡寄存器来计算、求值。因此,为了确保对寄存器有良好的分配,并取得较好的性能,应该尽量限制,使函数的内部循环最多只使用 12 个局部变量。

如果编译器确实需要替换变量,那么编译器将会根据变量的使用频度来选择要替换的变量。一个循环内的变量使用频度会高很多,所以可以按照变量所在循环层次来决定其重要性。一般最内层循环体内的变量就是最重要的。

在 C 语言中,关键词 register 表示编译器应该分配给指定变量一个寄存器。但是,不同的编译器对这个关键词的处理也不完全相同,不同的结构状态下可供分配的寄存器数目也不相同(比如 Thumb 和 ARM)。因此应尽量避免使用 register 关键词,而应依赖编译器正常的寄存器分配策略来分配。

小 结 高效的寄存器分配

- 应该尽量限制函数内部循环所用局部变量的数目,最多不超过 12 个,这样,编译器就可以把这些变量都分配给 ARM 寄存器;

- 可以引导编译器,通过查看是否属于最内层循环的变量来确定某个变量的重要性。

5.5 函数调用

ARM 过程调用标准(APCS)定义了如何通过寄存器传递函数参数和返回值。最近的 ARM - Thumb 过程调用标准(ATPCS)又增加了 ARM 和 Thumb 互相调用的说明。

函数中最前面的 4 个整型参数是通过 ARM 的前 4 个寄存器 r0, r1, r2 和 r3 来传递的。随后的整型参数是通过下降式满堆栈(full descending stack)来传递的,图 5.1 显示了参数的存放情况。函数返回的整型数据通过寄存器 r0 来传递。

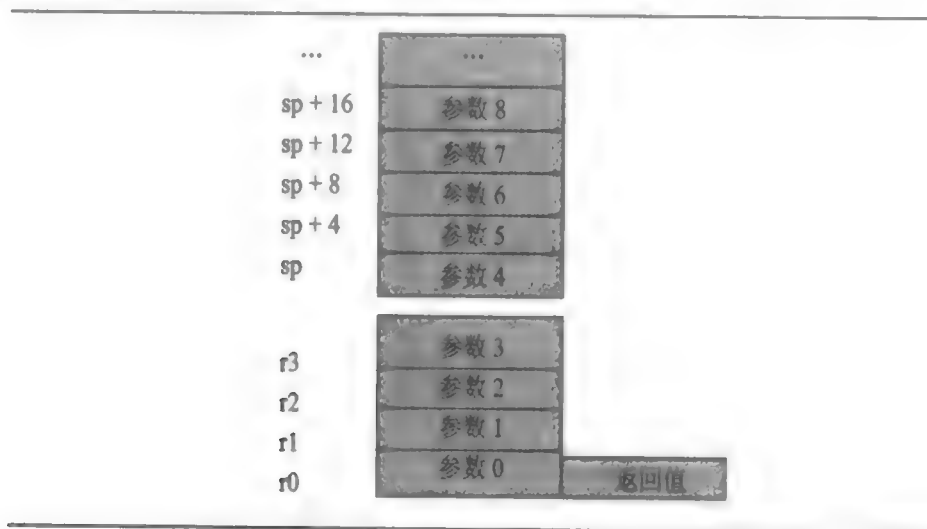


图 5.1 ATPCS 参数传递

这里的描述只针对整型和指针类型的参数。双字类型的参数,例如 long long 类型或者 double 类型,是通过一对连续的寄存器传递的,双字类型的返回值存放在 r0 和 r1 中。编译器可以通过寄存器或根据命令行选项参考来传递结构体。

首先应指出的是,过程调用标准是 4 寄存器规则(four - register rule)。带有 4 个或者更少参数的函数,要比多于 4 个参数的函数执行效率高得多。对带有少于 4 个参数的函数来说,编译器可以用寄存器传递所有的参数;而对于多于 4 个参数的函数,函数调用者和被调用者都必须通过访问堆栈来传递一些参数。

注意: 在 C++ 中,一个对象方法的第一个参数总是 this 指针。这个参数是隐式的,并附加到其它显式参数。

如果一个函数的参数多于 4 个,或者 C++ 中的显式参数多于 3 个,那么通常使用结构体,执行效率会更高。将多个相关的参数组织到一个结构体中,传递一个结构体指针来代替

多个参数。哪些参数应该归到一个结构体,这将依赖于应用程序。

下面的例子说明了使用结构体指针的好处。这是一个典型的子程序:从数组 data 插入 N 个字节到一个队列里面。这里用一个起始地址是 Q_start(包括此地址)、结束地址是 Q_end(不包括此地址)的循环缓冲区来实现这个队列。

```
char * queue_bytes_v1 (
    char * Q_start,      /* 队列缓冲区起始地址 */
    char * Q_end,        /* 队列缓冲区结束地址 */
    char * Q_ptr,        /* 当前队列指针位置 */
    char * data,          /* 插入队列的数据 */
    unsigned int N)      /* 插入数据的数目 */
{
    do
    {
        * (Q_ptr++) = * (data++);

        if (Q_ptr == Q_end)
        {
            Q_ptr = Q_start;
        }
    } while ( -- N );
    return Q_ptr;
}
```

编译后生成:

```
queue_bytes_v1
    STR    r14,[r13,# -4]!           ;save lr on the stack
    LDR    r12,[r13,# 4]             ;r12 = N
queue_v1_loop
    LDRB   r14,[r3],#1               ;r14 = * (data++)
    STRB   r14,[r2],#1               ; * (Q_ptr++) = r14
    CMP    r2,r1                     ;if (Q_ptr == Q_end)
    MOVEQ   r2,r0                    ;{ Q_ptr = Q_start;}
    SUBS   r12,r12,#1                ; -- N and set flags
    BNE    queue_v1_loop             ;if (N != 0) goto loop
    MOV    r0,r2                     ;r0 = Q_ptr
    LDR    pc,[r13],#4               ;return r0
```


可以将这个例子与使用 3 个函数参数的结构化的例子进行比较。

【例 5.6】 下面的代码定义了一个称为 Queue 的结构体,把这个结构体指针传递给函数,这样减少了函数参数的个数。

```
typedef struct {
    char * Q_start,    /* 队列缓冲区起始地址 */
    char * Q_end,      /* 队列缓冲区结束地址 */
    char * Q_ptr,      /* 当前队列指针位置 */
} Queue;

void queue_bytes_v1 (Queue * queue, char * data, unsigned int N)
{
    char * Q_ptr = queue->Q_ptr;
    char * Q_end = queue->Q_end;

    do
    {
        * (Q_ptr++) = * (data++);

        if (Q_ptr == Q_end)
        {
            Q_ptr = queue->Q_start;
        }
    } while ( --N );
    queue->Q_ptr = Q_ptr;
}
```

编译后生成:

```
queue_bytes_v2
    STR    r14,[r13,# -4]          ; save lr on the stack
    LDR    r3,[r0,# 8]            ; r3 = queue->Q_ptr
    LDR    r14,[r0,# 4]           ; r14 = queue->Q_end
queue_v2_loop
    LDRB   r12,[r1],#1            ; r12 = * (data++)
    STRB   r12,[r3],#1            ; * (Q_ptr++) = r12
    CMP    r3,r14                 ; if (Q_ptr == Q_end)
    LDREQ  r3,[r0,# 0]            ; Q_ptr = queue->Q_start
```

```

SUBS    r2,r2,#1                ; -- N and set flags
BNE     queue_v2_loop           ; if (N != 0) goto loop
STR     r3,[r0,#8]              ; queue->Q_ptr = r3
LDR     pc,[r13],#4             ; return

```

queue_bytes_v2 比 queue_bytes_v1 多了一条指令,但实际上总体效率却更高。相比前面的 5 个参数,在第 2 个例子中仅有 3 个函数参数,所以每次函数调用只须设置 3 个寄存器。而在第 1 个例子中,每次调用要有 4 个寄存器设置、一个压栈、一个退栈。在函数调用开销方面,净省了 2 条指令。对被调用函数来说,可以节省更多,因为它只须分配一个寄存器给 queue 结构指针,而不需要像前面非结构化的例子中那样分配 3 个寄存器。

如果函数体很小,只用到很少的寄存器(很少的局部变量),那么还有一些其它的方法来减小函数调用的开销。可以把调用函数和被调用函数放在同一个 C 文件中,这样编译器就知道了被调用函数生成的代码,并以此对调用函数进行一些优化:

- 调用函数不需要保护被调用函数没有用到的寄存器,因此,调用函数也不需要保护所有 ATPCS 占用的寄存器;
- 如果被调用的函数很小,那么编译器可以在调用函数中内联被调用函数的代码,这样可以彻底去除函数调用开销。

【例 5.7】 函数 uint_to_hex 把一个 32 位的无符号整数转换为 8 个十六进制数。

这个函数调用了辅助函数 nybble_to_hex,把一个在范围 0~15 的数字 d 转换为一个十六进制数字。

```

unsigned int nybble_to_hex ( unsigned int d )
{
    if ( d<10 )
    {
        return d + '0';
    }
    return d - 10 + 'A';
}

void uint_to_hex ( char * out,unsigned int in )
{
    unsigned int i;

    for ( i=8;i!=0;i-- )
    {
        in = ( in << 4 ) | ( in >> 28 ); /* 循环左移 4 位 */
    }
}

```

```

        * (out++) = (char) nybble_to_hex (in & 15);
    }
}

```

编译后, 可以看到函数 `uint_to_hex` 根本没有调用函数 `nybble_to_hex`! 在下面编译后的代码中, 编译器内联了 `uint_to_hex` 的代码。这将比函数调用效率高得多。

```

uint_to_hex
    MOV        r3, #8                      ;i = 8
uint_to_hex_loop
    MOV        r1,r1,ROR #28                ;in = ( in << 4 ) | ( in >> 28)
    AND        r2,r1,#0xf                  ;r2 = in & 15
    CMP        r2,#0xa                      ;if ( r2 >= 10)
    ADDCS      r2,r2,#0x37                   ;    r2 += 'A' - 10
    ADDCC      r2,r2,#0x30                   ;else r2 += 10
    STRB       r2,[r0],#1                   ; * (out++) = r2
    SUBS       r3,r3,#1                     ;i -- and set flags
    BNE        uint_to_hex_loop              ;if ( i!= 0) goto loop
    MOV        pc,r14                       ;return

```

编译器只会内联比较小的函数。也可以使用 `_inline` 关键字要求编译器内联一个函数, 不过这个关键字只是一个提示, 编译器可能会忽略它(关于内联函数的更多信息可参见 5.12 节)。内联一个大的函数将会大大增加代码量, 而对性能却不会有大的改善。

117

小结 高效地调用函数

- 尽量限制函数的参数, 不要超过 4 个, 这样函数调用的效率会更高。也可以将几个相关的参数组织在一个结构体中, 用传递结构体指针来代替多个参数。
- 把比较小的被调用函数和调用函数放在同一个源文件中, 并且要先定义, 后调用, 编译器就可以优化函数调用或者内联较小的函数。
- 对性能影响较大的重要函数可使用关键字 `_inline` 进行内联。

5.6 指针别名

当 2 个指针指向同一个地址对象时, 这 2 个指针被称作该对象的别名(alias)。如果对其中一个指针进行写入, 就会影响从另一个指针的读出。在一个函数中, 编译器通常不知道哪一个指针是别名, 哪一个不是; 或哪一个指针有别名, 哪一个没有。编译器必须非常悲观的认为, 对任何一个指针的写入, 都将会影响从任何其它指针的读出, 但这样会明显降低代

ARM 嵌入式系统开发

码执行的效率。

先看一个非常简单的例子。下面的函数对 2 个定时器值使用一个步进量进行累加：

```
void timer_v1 ( int * timer1,int * timer2,int * step )
{
    * timer1 += * step ;
    * timer2 += * step ;
}
```

编译后生成：

```
timers_v1
    LDR        r3,[r0,# 0]                ;r3 = * timer1
    LDR        r12,[r2,# 0]              ;r12 = * step
    ADD        r3,r3,r12                  ;r3 += r12
    STR        r3,[r0,# 0]                ; * timer1 = r3
    LDR        r0,[r1,# 0]                ;r0 = * timer2
    LDR        r2,[r2,# 0]                ;r2 = * step
    ADD        r0,r0,r2                    ;r0 += r2
    STR        r0,[r1,# 0]                ; * timer2 = r0
    MOV        pc ,r14                    ;return
```

注意编译器装载 step 两次。通常，一种被称为公共子表达式消除(common subexpression elimination)的编译器选项，可以使编译器优化 * step，只被装载一次；第二次使用时，其值将会被重复使用。但是，在这里编译器不能使用这种优化。指针 timer1 和指针 step 可能会互为别名。换句话说，编译器不能确定对指针 timer1 的写入是否会影响从指针 step 的读出。在这种情况下，第二次 * step 的值将与第一次的不同，将会是 * timer1 的值。这就使编译器不得不增加一条额外的 load 指令。

如果使用结构体来代替直接的指针访问，也会出现同样的问题。下面的代码编译后的结果执行效率也不高：

```
typedef struct { int step ; } State ;
typedef struct { int timer1,timer2 ; } Timers;

void timers_v2 ( State * state, Timers * timers )
{
    timers -> timer1 += state -> step ;
    timers -> timer2 += state -> step ;
}
```

为了防止 `state->step` 和 `timers->timer1` 处在同一个存储地址,编译器必须对 `state->step` 求值 2 次。若想避免这种情况也是容易的:定义一个新的局部变量来保存 `state->step` 的值。这样编译器就只进行一次装载了。

【例 5.8】 在 `timers_v3` 代码中,使用一个局部变量 `step` 来保存 `state->step` 的值。现在编译器就不需要担心 `state` 和 `timers` 的别名问题了。

```
void timers_v3 ( state * state, Timers * timers )
{
    int step = state->step ;

    timers->timer1 += step ;
    timers->timer2 += step ;
}
```

另外,对其它一些不明显的别名情况也要小心。当调用其它函数时,被调用的函数可能会改变存储器的内容,这样也就可能改变了所有涉及存储器读的表达式值,编译器将会重新对相关表达式进行求值。例如,假设先读 `state->step`,调用一个函数后,再读 `state->step`。编译器必须假定调用的函数会改变内存中 `state->step` 的值,因此就会执行 2 次读操作,而不是再次使用第一次读到的 `state->step` 的值。

另一个问题,是要获取局部变量的地址。一旦这么做了,这个变量就被一个指针所对应,就可能与其它指针产生别名。万一别名发生,编译器宁可堆栈重新读入数据。考虑下面的例子,读入一个数据包并计算它们的校验和:

```
int checksum_next_packet (void)
{
    int *data ;
    int N, sum = 0;
    data = get_next_packet (&N);

    do
    {
        sum += * (data ++);
    } while ( -- N );

    return sum;
}
```

这里 `get_next_packet` 函数返回下一个数据包的地址和大小。下面是编译后生成的代码:

```

checksum_next_packet
    STMFD    r13!, {r4, r14}      ;save r4,lr on the stack
    SUB      r13,r13,#8           ;create two stacked variables
    ADD      r0,r13,#4            ;ro = &N,N stacked
    MOV      r4,#0                ;sum = 0
    BL       get_next_packet      ;r0 = data
checksum_loop
    LDR      r1,[r0],#4           ;r1 = * (data ++ )
    ADD      r4,r1,r4             ;sum += r1
    LDR      r1,[r13,#4]          ;r1 = N (read from stack )
    SUBS     r1,r1,#1             ;r1 -- & set flags
    STR      r1,[r13,#4]          ;N = r1 (write to stack )
    BNE      checksum_loop        ;if ( N != 0 ) goto loop
    MOV      r0,r4                ;r0 = sum
    ADD      r13,r13,#8           ;delete stacked variables
    LDMFD    r13!,{ r4,pc }       ;return r0

```

注意对于每一次 N 编译器是如何从堆栈读/写 N 的。一旦得到了 N 的地址,并将它传递给 `get_next_packet`,编译器就需要担心别名问题,因为指针 `data` 和 `&N` 可能会是别名。为了避免这种情况,不要使用局部变量的地址。如果必须这样做,那么可以在使用之前先把它的值复制到另外一个局部变量。

读者可能会疑惑,为什么编译器要分配 2 个堆栈变量的空间而实际只用 1 个。这是为了保持堆栈的 8 字节边界对齐,在 ARMv5TE 结构体系中的 LDRD 指令要求这样做。上面的例子实际上没有使用 LDRD,但是编译器不知道 `get_next_packet` 是否会使用这条指令。

小结 避免指针别名

- 不要依赖编译器来消除包含存储器访问的公共子表达式,而应建立一个新的局部变量来保存这个表达式的值,这样可以保证只对这个表达式求一次值;
- 避免使用局部变量的地址,否则对这个变量的访问效率会比较低。

5.7 结构体安排

通常,使用结构体是因为觉得结构体可明显地改善性能和增强代码密度。在 ARM 上使用结构体有 2 个问题需要考虑:结构体地址边界对齐和结构体总的大小。

对于 ARMv5TE 及其以上的体系结构,load 和 store 指令仅仅保证从与访问宽度尺寸对齐的地址来装载和存储数据。表 5.4 列出了这些限制。

基于这个原因,ARM 编译器将会自动把一个结构体的起始地址与该结构体中最大访问数据宽度(通常 4 或者 8 字节)的倍数对齐,并通过插入填充位把结构体地址边界与它们的存取宽度相对齐。

例如,看下面的结构体:

```
struct {
    char a;
    int b;
    char c;
    short d;
}
```

对于小端(little-endian)存储器系统,编译器会增加填充位(pad)安排数据,以确保下一个目标与其尺寸要求的地址相对齐:

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

表 5.4 ARMv5TE 中 load 和 store 的边界限制

传递的字节数	指令	字节地址
1 byte	LDRB, LDRSB, STRB	任何字节地址对齐
2 bytes	LDRH, LDRSH, STRH	2 字节的倍数地址对齐
4 bytes	LDR, STR	4 字节的倍数地址对齐
8 bytes	LDRD, STRD	8 字节的倍数地址对齐

为了提高存储器的空间利用率,应该重新安排各变量元素的位置:

```
struct {
    char a;
    char c;
    short d;
    int b;
}
```

这样,结构体的大小从 12 字节减少到 8 字节,下面是新的编排:

Address	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

因此,这是组织一个结构体内各元素的好方法,这样,结构体存储时就不需要插入不必要的填充位。armcc 编译器支持一个关键字 `__packed`,表示去除所有的填充位。例如,结构体

```
__packed struct{
    char a;
    int b;
    char c;
    short d;
}
```

在内存中将会安排为

Address	+3	+2	+1	+0
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	d[15,8]	d[7,0]	c	b[31,24]

这样,存储空间是不浪费了,但访问 packed 结构体的速度慢了,而且效率比较低。由于数据边界不对齐,编译器只能通过多个边界对齐的操作,把结果合并、重组,来模拟边界不对齐的 load 和 store 操作。所以,一般只有在程序的代码空间比执行效率更重要,而且重新排列并不能减少填充位的情况下,才使用 `__packed` 关键字。还有就是在移植一个在存储器中有确定结构安排的代码时,也会用到它。

在存储器中对一个结构体的确切安排依赖于编译器供应商和所使用的编译器版本。在 API(Application Programmer Interface)定义中,人工把不能去除的填充位插入结构体中,不失为一种好方法。这种明确的结构体安排方法,对连接不同供应商和版本的编译器生成的代码是有利的。

另一个不明确的问题是枚举(enum)类型。不同的编译器根据枚举的范围,对于枚举类型会分配不同的空间大小。例如,考虑类型

```
typedef enum {
    FALSE,
    TRUE
} Bool ;
```


在 ADS1.1 中的 armcc 编译器认为 Bool 是 1 字节的数据类型,因为 Bool 只使用值 0 和 1。在一个结构体中,Bool 仅仅占用 8 位的空间。但是,gcc 认为 Bool 是一个字类型,在一个结构体中会占用 32 位的空间。为了避免这种不确定性,最好在 API 使用的结构体中避免使用 enum 类型。

另一个须考虑的问题是结构体的尺寸和结构体中各元素的偏移量,这是一个在 Thumb 指令集下编译时最为敏感的问题。Thumb 指令只有 16 位宽度,因此从一个结构体基地址指针起,只允许较小的元素偏移量。表 5.5 显示了在 Thumb 方式下装载和存储基寄存器的偏移量。

表 5.5 Thumb 方式下 load 和 store 的偏移量

指 令	从基寄存器起的有效偏移量
LDRB, LDRSB, STRB	0~31 个字节
LDRH, LDRSH, STRH	0~31 个半字(0~62 字节)
LDR, STR	0~31 个字 (0~124 字节)

因此编译器产生一条指令,只能访问存放在结构体前 32 字节中的一个 8 位大小的结构体元素。类似,单一指令只能访问存在于结构体前 64 字节中的一个 16 位大小的元素,或存在于结构体前 128 字节中的一个 32 位大小的元素。一旦超过了这个限制,对结构体访问的效率将会变得很低。

遵循下面的规则来组织元素构造一个结构体,可以获得最高的效率:

- 把所有 8 位大小的元素安排在结构体的前面;
- 依次安排 16 位、32 位和 64 位的元素;
- 把所有数组和比较大的元素安排在结构体的最后;
- 对于一条指令,如果结构体太大而不能访问所有的元素,那么把元素组织到一个子结构体中。编译器可以维持单独的子结构体的指针。

小 结 高效的 结构体 安排

- 结构体元素要按照元素的大小来排列,以最小的元素放在开始,最大的元素安排在最后;
- 避免使用很大的结构体,可以用层次化的小结构体来代替;
- 为了提高可移植性,人工对 API 的结构体增加填充位,这样,结构体的安排将不会依赖于编译器;
- 在 API 的结构体中要谨慎使用枚举(enum)类型。一个枚举类型的大小是编译器相关的。

5.8 位 域

位域可能是在 ANSI C 规范中最小的标准化部分了。位域被声明用来存放特定数目的位。编译器可以选择各个位的分配位置,单单就这个原因,应该避免在一个联合(union)或者 API 结构体定义中使用位域。不同的编译器可以对相同的位域分配不同的位置。

避免使用位域对提高效率是有好处的。位域是结构体的元素,通常使用结构体指针进行访问;所以,它们也同样存在着 5.6 节中提到的指针别名问题。每一个对位域的访问实际上就是对存储器的访问,可能出现的指针别名问题使得编译器需要经常重新装载位域。

下面的例子,dostages_v1,举例说明了这个问题。同时也显示了编译器不会很好地倾向于去优化位域测试。

```
void dostageA (void );
void dostageB (void );
void dostageC (void );

typedef struct {
    unsigned int stageA ,1;
    unsigned int stageB ,1;
    unsigned int stageC ,1;
} Stages_v1 ;

void dostages_v1(Stages_v1 * stages )
{
    if ( stages ->stageA)
    {
        dostageA () ;
    }
    if ( stages ->stageB)
    {
        dostageB () ;
    }
    if ( stages ->stageC)
    {
        dostageC () ;
    }
}
```

这里使用了 3 个位标志来表示处理的 3 个不同阶段。这个例子编译后生成：

```
dostages_v1
    STMFD      r13!,{ r4,r14 }      ;stack  r4.lr
    MOV        r4,r0                ;move stages to r4
    LDR        r0,[r0,#0]            ;r0 = stage bitfield
    TST        r0,#1                 ;if (stage -> stageA)
    BLNE       dostageA              ; { dostageA();}
    LDR        r0,[r4,#0]            ;r0 = stages bitfield
    MOV        r0,r0,LSL #30         ;shift bit 1 to bit 31
    CMP        r0,#0                 ;if (bit31)
    BLT        dostageB              ; { dostageB();}
    LDR        r0,[r4,#0]            ;r0 = stages bitfield
    MOV        r0,r0,LSL #29         ;shift bit 2 to bit 31
    CMP        r0,#0                 ;if (! bit31)
    LDMLTFD    r13!,{ r4,r14 }      ; return
    BLT        dostageC              ;dostageC();
    LDMFD      r13!,{ r4,pc }        ;return
```

注意编译器一共访问了包含位域的存储器位置 3 次, 因为位域存储在存储器中, dostage 函数可能会改变值。另外, 编译器使用两条指令来测试位域中的 bit 1 和 bit 2。

使用一个整型数来代替一个位域, 可以产生效率高得多的代码。使用 enum 或者 #define 屏蔽来把一个整型数分成几个不同的域。

【例 5.9】 下面的代码使用逻辑操作而不是位域来实现 dostages 函数：

```
typedef unsigned long Stages_v2
#define STAGEA ( 1ul << 0 )
#define STAGEB ( 1ul << 1 )
#define STAGEC ( 1ul << 2 )

void dostages_v2(Stages_v2 * stages_v2 )
{
    Stages_v2  stages = * stages_v2 ;

    if ( stages & STAGEA)
    {
        dostageA ();
    }
}
```

```

    if ( stages & STAGEB )
    {
        dostageB ( ) ;
    }

    if ( stages & STAGEC )
    {
        dostageC ( ) ;
    }
}

```

既然一个 unsigned long 类型包含了所有的位域,那么就可以把这些位域的值保存到一个局部变量 stages,这可以避免在 5.6 节中讨论的内存变量别名问题。换句话说,编译器必须假定 dostageX(这里 X 表示 A,B 或者 C)函数可能会改变 * stages_v2 的值。

编译器生成下面的代码,代码量比前面使用 ANSI 位域的版本减少了 33%:

```

dostages_v2
    STMFED        r13!,{ r4,r14 }        ;stack  r4,lr
    LDR           r4,[r0,#0]              ;stage = * stages_v2
    TST           r4,#1                   ;if (stage & STAGEA )
    BLNE          dostageA                 ; { dostageA ();}
    TST           r4,#2                   ;if (stage & STAGEB )
    BLNE          dostageB                 ; { dostageB();}
    TST           r4,#4                   ;if (! (stages & STAGEC))
    LDMNEFD       r13 !,{ r4,r14 }        ;return
    BNE           dostageC                 ;dostageC ();
    LDMFD         r13!,{ r4,pc }          ;return

```

也可以使用屏蔽位来设置和清除位域,这和测试位域一样简单。下面的代码显示了如何使用 STAGE 屏蔽位来设置、清除或者取反位:

```

stages |= STAGEA ;                /* enable stage A */
stages &= ~STAGEB ;                /* disable stage B */
stages ^= STAGEC ;                /* toggle stage C */

```

这几个位的设置、清除和取反操作都只使用了一条 ARM 指令,分别为 ORR,BIC 和 EOR 指令。另外还可以使用一条指令同时处理几个位域。例如:

```

stages |= (STAGEA | STAGEB) ;      /* enable stages A and B */
stages &= ~( STAGEA | STAGEC );    /* disable stages A and C */

```

小结 位域

- 应避免使用位域, 而使用 `#define` 或者 `enum` 来定义屏蔽位;
- 使用整型逻辑运算 AND、OR、“异或”操作和屏蔽对位域进行测试、取反和设置操作。这些操作编译效率高, 还可以同时对多个位域进行测试、取反和设置。

5.9 边界不对齐数据和字节排列方式(大/小端)

边界不对齐数据和字节排列方式这 2 个问题, 可使内存访问和移植问题复杂化。须考虑数组指针是否边界对齐, ARM 配置是大端(big-endian), 还是小端(little-endian)的存储器系统。

ARM 的 load 和 store 指令假定地址是正在装载或者存储的数据类型长度的倍数。如果装载和存储的地址与数据类型边界不对齐, 那么可能会产生异常的结果, 例如数据异常或者装载了一个错位的值。对于质量好、可移植的代码, 应尽量避免使用边界不对齐的访问。

通常情况下 C 编译器假定指针是边界对齐的。如果指针不是边界对齐的, 那么程序的执行会产生不正确的结果。这样, 把代码从那些允许边界不对齐访问的处理器移植到 ARM 时就会出现异常。对 armcc 来说, 保留字 `__packed` 告诉编译器一个数据可以放在任意字节对齐的位置。这对移植代码是很有用的, 但显然会影响性能。

为了说明这个问题, 看下面的简单例子 `readint`。这个函数返回由 `data` 指向地址的一个整数, 并且已经使用了 `__packed` 告诉编译器: 这个整数可能不是边界对齐的。

```
int readint ( __packed int * data )
{
    return * data ;
}
```

编译后生成:

```
readint
    BIC        r3,r0,#3           ;r3 = data & 0xffffffffc
    AND        r0,r0,#3           ;r0 = data & 0x00000003
    MOV        r0,r0,LSL #3       ;r0 = bit offset of data word
    LDMIA      r3,{ r3,r12 }      ;r3,r12 = 8 bytes read from r3
    MOV        r3,r3,LSR r0       ;these three instructions
    RSB        r0,r0,#0x20        ;shift the 64 bit value r12, r3
    ORR        r0,r3,r12,LSL r0   ;right by r0 bits
    MOV        pc,r14            ;return r0
```

ARM 嵌入式系统开发

注意这段代码的大小和复杂度。编译器使用 2 个边界对齐的访问和数据处理操作来仿真边界不对齐的访问,开销很大,并说明了为什么要避免使用 `__packed`。替代的做法是,使用类型 `char *` 来指向可以出现在任何边界的数据,这样就不会出现不对齐的访问。后面将介绍从一个 `char *` 指针读 32 位字的更高效的方法。

通常计算机之间传输信息,在读数据包或者文件时,就会碰到边界对齐问题。网络报文和压缩过的图像文件就是很好的例子。在这些文件中,2 字节或 4 字节的整数可能会出现在任意的偏移量位置。数据已尽可能地被压缩,损害了边界对齐。

从数据包或压缩过的文件中读数据时,数据字节排列方式也是一个很大的问题。ARM 核可以被配置成小端(little-endian,最低字节在最低地址)或者大端(big-endian,最高字节在最低地址)工作模式。通常情况下默认是小端(little-endian)模式。

ARM 的字节排列方式通常在电源开启时设置,以后一直保持不变。表 5.6 和 5.7 举例说明了 ARM 中 8 位、16 位和 32 位的 load 和 store 指令对于不同的字节排序方式是如何工作的。这里假定字节地址 A 与存储器传输的大小是对齐的。表格显示了存储器中字节方式的地址如何映射到 load 和 store 指令操作的 32 位寄存器。

表 5.6 小端(little-endian) 配置

指令	宽度/bits	b31~b24	b23~b16	b15~b8	b7~b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRB	8	X	X	X	B(A)
LDRH	16	0	0	B(A+1)	B(A)
LDRSH	16	S(A+1)	S(A+1)	B(A+1)	B(A)
STRH	16	X	X	B(A+1)	B(A)
LDR/STR	32	B(A+3)	B(A+2)	B(A+1)	B(A)

表 5.7 大端(big-endian) 配置

指令	宽度/bits	b31~b24	b23~b16	b15~b8	b7~b0
LDRB	8	0	0	0	B(A) ^①
LDRSB	8	S(A)	S(A)	S(A) ^②	B(A)
STRB	8	X	X	X ^③	B(A)
LDRH	16	0	0	B(A)	B(A+1)
LDRSH	16	S(A)	S(A)	B(A)	B(A+1)
STRH	16	X	X	B(A)	B(A+1)
LDR/STR	32	B(A)	B(A+1)	B(A+2)	B(A+3)

注: ① 地址 A 处的字节数据;

② B(A)的符号位,如果 B(A)的位 7 是 1,则 S(A)为 0xff;否则为 0x00;

③ 这些位在写操作时是被忽略的。

什么是处理字节排列方式和边界对齐问题的最好方法呢？对执行速度要求不是很严格的程序，用例子 5.10 中的函数 `readint_little` 和 `readint_big` 来说明这个问题。该例从边界可能不对齐的存储器中读一个 4 字节的整数。地址边界是否对齐在程序编译时刻是未知的，只有到运行时才知道。如果已经装载了一个大端数据的文件，比如 JPEG 图像，那么就用 `readint_big`；而对小端数据的字节流，就调用 `read_little`。如果不考虑 ARM 配置的存储器字节排序方式，那么 2 个函数都能正确地执行。

【例 5.10】 从一个 `data` 指向的字节流读一个 32 位整型数据的函数。

字节流可以分别包含小端或大端数据。这 2 个函数与 ARM 配置的存储器系统字节排列方式无关，因为它们只用字节访问。

```
int readint_little(char *data)
{
    int a0,a1, a2,a3;
    a0 = *(data++);
    a1 = *(data++);
    a2 = *(data++);
    a3 = *(data++);
    return a0 | (a1 << 8) | (a2 << 16) | (a3 << 24);
}

int readint_big(char *data)
{
    int a0,a1, a2,a3;
    a0 = *(data++);
    a1 = *(data++);
    a2 = *(data++);
    a3 = *(data++);
    return (((a0 << 8) | a1) << 8) | a2 << 16 | a3;
}
```

如果对程序执行速度有严格要求，那么最便捷的方法是写几个关键程序的变体，对每种可能的边界对齐和 ARM 字节排序方式编写不同的程序段。这样就可以根据实际情况调用不同的已经优化过的程序段。

【例 5.11】 `read_samples` 程序处理从一个以地址 `in` 起始的 N 个 16 位的音频采样数据。这个音频采样数据是小端的（例如从 `a.wav` 文件），而且是字节对齐的。程序把采样数据复制到以 `out` 指向的 `short` 类型的数组。这些采样数据将会根据 ARM 内存的字节排序

方式来存储。

这个程序段高效地处理了所有可能出现的情况,也不管输入边界对齐和 ARM 字节排序方式的配置。

```
void read_samples ( short * out, char * in, unsigned int N )
{
    unsigned short * data ;    /* aligned input pointer */
    unsigned int sample, next ;

    switch ( ( unsigned int ) in & 1 )
    {
        case 0 : /* the input pointer is aligned */
            data = ( unsigned short * ) in ;
            do
            {
                sample = * ( data ++ ) ;
                # ifdef __BIG_ENDIAN
                    sample = ( sample >> 8 ) | ( sample << 8 ) ;
                # endif
                * ( out ++ ) = ( short ) sample ;
            } while ( -- N ) ;
            break ;

            case 1 : /* the input pointer is not aligned */
                data = ( unsigned short * ) ( in - 1 ) ;
                sample = * ( data ++ ) ;
                # ifdef __BIG_ENDIAN
                    sample = sample & 0xff ; /* get first byte of sample */
                # else
                    sample = sample >> 8 ; /* get first byte of sample */
                # endif
                do
                {
                    next = ( data ++ ) ;
                    /* complete one sample and start the next */
                    # ifdef __BIG_ENDIAN
                        * out ++ = ( short ) ( ( next & 0xff00 ) | sample ) ;
                        sample = next & 0xff ;
```



```

    #else
        *out++ = (short)((next << 8) | sample);
        sample = next >> 8;
    #endif

    } while( --N );
    break ;
}
)

```

整个程序对每一种字节排列方式和边界对齐给出了不同的代码段。字节排列方式在编译时用 `__BIG_ENDIAN` 编译器标志表示。边界对齐在运行时使用 `switch` 状态语句来分别处理。

为进一步提高程序执行效率,甚至可以用 32 位读/写来代替 16 位读/写,在 `switch` 语句中产生 4 个分支,每个分支处理一种可能的地址边界对齐问题。

小 结 字节排列方式和边界对齐

- 尽量避免使用边界不对齐的数据;
- 使用类型 `char *` 可指向任意字节边界的数据。通过读字节来访问数据,使用逻辑操作来组合数据,这样代码就不会依赖于边界是否对齐或者 ARM 的字节排列方式的配置;
- 为了快速访问边界不对齐的结构体,可以根据指针边界和处理器的字节排序方式写出不同的程序变体。

131

5.10 除 法

ARM 硬件上不支持除法指令。编译器是通过调用 C 库函数来实现除法运算的。有许多不同类型的除法程序来适应不同的除数和被除数。第 7 章将着重分析汇编除法程序。C 库函数中的标准整数除法程序,根据执行情况和输入操作数的范围,要花费 20~100 个周期。

除法和模运算(`/`和`%`)执行起来比较慢,所以应该尽可能地避免使用。但是,除数是常数的除法运算和用同一个除数的重复除法,执行效率会比较高。本节描述了如何用乘法运算代替除法运算,以及如何使除法调用的次数最少化。

对环形缓冲区操作经常要用到除法,其实完全可以避免这些除法运算。假定有一个 `buff_size` 大小的环形缓冲区, `offset` 指定目前所在的位置。通过 `increment` 字节来增加 `off-`

ARM 嵌入式系统开发

set 的值,一般是这样写的:

```
offset = ( offset + increment ) % buffer_size;
```

效率更高的写法是:

```
offset += increment ;
if ( offset >= buffer_size )
{
    offset -= buffer_size;
}
```

第一种写法须花费 50 个周期,而第二种因为没有除法运算,只须花费 3 个周期。这里假定 $\text{increment} < \text{buffer_size}$,在实际应用中这点应该是保证的。

如果不能避免除法运算,那么就应尽量使除数和被除数是无符号的整数。有符号的除法程序执行起来更加慢,因为它们先要取得除数和被除数的绝对值,再调用无符号除法运算,最后再确定结果的符号。

许多 C 语言库中的除法函数返回商和余数。换句话说,每一个除法运算,余数是可以无偿得到的,反之亦然。例如,要在屏幕缓冲区找到偏移量为 offset 的屏幕位置(x,y),可以这样写:

```
typedef struct {
    int x ;
    int y;
}point ;

point getxy_v1 ( unsigned int offset,unsigned int bytes_per_line )
{
    point p ;
    p.y = offset/bytes_per_line ;
    p.x = offset - p.y * bytes_per_line ;
    return p ;
}
```

这里,似乎对 p.x 使用减法和乘法,少了一次除法运算;但是,实际上使用模运算或者取余操作效率更高。

【例 5.12】 在 getxy_v2 中,对除法程序,商和余数操作只需要一次调用。

```
point getxy_v2 ( unsigned int offset,unsigned int bytes_per_line )
{
```

```

point p;

p.x = offset % bytes_per_line ;
p.y = offset/bytes_per_line ;
return p ;
}

```

从下面编译器的输出结果可以看到,只有一次除法调用。实际上,这个程序要比前面的 `getxy_v1` 少 4 条指令。

注意: 并不是对所有的编译器和 C 库都有这样的结果。

```

getxy_v2
      STMFID          r13!,{ r4,r14 }      ;stack  r4,lr
      MOV             r4,r0                ;move p to r4
      MOV             r0,r2                ;r0 = bytes_per_line
      BL              __rt_udiv             ;( r0,r1 ) = ( r1/r0, r1 % r0)
      STR             r0,[r4,#4]           ;p.y = offset/bytes_per_line
      STR             r1,[r4,#0]           ;p.x = offset % bytes_per_line
      LDMFD           r13!,{ r4,pc}        ;return

```

5.10.1 带余数的无符号重复除法

在程序中,同一个除数的除法经常会出现很多次。在前面的例子中, `bytes_per_line` 的值在整个程序中都是固定不变的。如果使用 3 到 2 笛卡尔坐标表示,那么就可以使用同一个除数两次:

$$(x, y, z) \rightarrow (x/z, y/z)$$

这种情况下,使用 `cache` 指令中的值 $1/z$, 并使用 $1/z$ 的乘法来代替除法运算,效率会更高。如何实现,这个问题将在下一小节讨论。另外,要尽可能使用 `int` 类型的运算,避免使用浮点运算。

下面的描述将更加偏重于从数学和理论的角度分析,把重复除法转换成乘法运算。如果读者对此没有兴趣,可以直接跳到后面的例 5.13。

5.10.2 把除转换为乘

将使用下面的符号,来区分精确的数学意义上的除法和整型除法运算。

- n/d = 整数 n 被分成 d 份, 结果趋向于 0 (与 C 语言相同);
- $n \% d = n$ 被 d 除之后的余数, 就是 $n - d (n/d)$;
- $\frac{n}{d} = nd^{-1}$ = 真正的数学意义上的 n 被 d 除。

当使用整型除法时, 最容易估算 d^{-1} 的值的办法是计算 $2^{32}/d$ 。然后, 就可以估算 n/d

$$(n (2^{32}/d)) / 2^{32} \quad (5.1)$$

在执行 n 的乘法时, 需要精确到 64 位。对于这种方法, 会出现如下问题:

- 为了计算 $2^{32}/d$, 由于一个 unsigned int 类型的数据放不下 2^{32} , 编译器须使用 64 位的 long long 类型的数, 而且必须指定除法为 $(1ull \ll 32)/d$ 。这种 64 位的除法比 32 位的除法执行起来要慢得多。
- 如果 d 碰巧是 1, 那么 $2^{32}/d$ 就不再适合于 unsigned int 数据类型。

上面的做法似乎很好, 而且解决了这 2 个问题。那么, 我们再来看一下用 $(2^{32}-1)/d$ 代替 $2^{32}/d$ 。

让

$$s = 0xffffffff \text{ ul} / d; \quad /* s = (2^{32} - 1) / d */$$

可以使用一个 unsigned int 类型除法来计算 s 。我们知道

$$2^{32} - 1 = sd + t, (0 \leq t < d) \quad (5.2)$$

所以

$$s = \frac{2^{32}}{d} - e_1, \quad \text{这里 } 0 < e_1 = \frac{1+t}{d} \leq 1 \quad (5.3)$$

接着, 计算 n/d 的值 q :

$$q = (\text{unsigned int}) ((\text{unsigned long long}) n * s) \gg 32$$

数学上, 32 位的右移会带来一个错误 e_2 :

$$q = ns2^{-32} - e_2 \quad \text{对于 } 0 \leq e_2 < 1 \quad (5.4)$$

替换 s 为:

$$q = \frac{n}{d} - ne_1 2^{-32} - e_2 \quad (5.5)$$

所以, q 是小于 n/d 的, 现在

$$0 \leq ne_1 2^{-32} + e_2 < e_1 + e_2 < 2 \quad (5.6)$$

因此

$$n/d - 2 < q \leq n/d \quad (5.7)$$

所以, $q = n/d$ 或 $q = (n/d) - 1$ 。可以发现, 通过计算余数 $r = n - qd$ ($0 \leq r < 2d$) 是比较容易的。下面的代码纠正了这个结果:

```
r = n - q * d; /* the remainder in the range 0 ≤ r < 2 * d */
if ( r ≥ d ) /* if correction is required */
{
    r -= d; /* correct the remainder to the range 0 ≤ r < d */
    q ++; /* correct the quotient */
}
/* now q = n/d and r = n % d */
```

【例 5.13】 显示实际上除法是如何转化为乘法的。

这个程序完成了 N 个元素的数组被 d 除这一任务。首先, 计算上面所说的 s 值, 然后用乘以 s 来代替每个被 d 除的除法。64 位的乘是很容易实现的, 因为 ARM 中有一条指令 UMULL, 可以进行 2 个 32 位数相乘, 给出一个 64 位的结果。

```
void scale (
    unsigned int * dest; /* 目的数据 */
    unsigned int * src; /* 源数据 */
    unsigned int d; /* 分母 d */
    unsigned int N;) /* 数据长度 */
{
    unsigned int s = 0xFFFFFFFF/d;
    do
    {
        unsigned int n, q, r;
        n = * (src ++ );
        q = (unsigned int) (((unsigned long long) n * s) >> 32);
        r = n - q * d;
        if ( r ≥ d ) /* if correction is required */
        {
            q ++;
        }
        * (dest ++ ) = q;
    } while ( -- N );
}
```

这里假定除数和被除数都是 32 位的无符号整数。当然,使用 32 位乘法进行 16 位的无符号数计算,或者使用 128 位乘法进行 64 位数计算,运算规则是一样的。可以为特定的数据选择最窄的运算宽度。如果数据是 16 位的,那么就设置 $s=(2^{16}-1)/d$,然后用标准的整型乘法来求值 q 。

5.10.3 除数是常数的无符号除法

如果除数是常数 c ,那么就可以使用例 5.13 的做法,先计算 $s=(2^{32}-1)/c$ 。但是,这里有一个更加有效的方法。ADS1.2 编译器使用这种方法来合成除数是常数的除法。

这种方法是使用一个足够精确的接近于 d^{-1} 的数,这样就可以给出精确的值 n/d 。这里使用下面的数学结论(对下面的第一个结论,可参考论文:Torbjorn Granlund and Peter L. Montgomery, "Division by Invariant Integers Using Multiplication", in proceedings of the SIG-PLAN PLDI'94 Conference, June 1994.);

如果 $2^{N+k} \leq ds \leq 2^{N+k} + 2^k$, 那么 $n/d = (ns) \gg (N+k)$, $(0 \leq n < 2^N)$ (5.8)

如果 $2^{N+k} - 2^k \leq ds \leq 2^{N+k}$, 那么 $n/d = (ns + s) \gg (N+k)$, $(0 \leq n < 2^N)$ (5.9)

因为 $n = (n/d)d + r$, $(0 \leq r \leq d-1)$

可得 $ns - (n/d)2^{N+k} = ns - \frac{n-r}{d}2^{N+k} = n \frac{ds - 2^{N+k}}{d} + \frac{r2^{N+k}}{d}$ (5.10)

$(n+1)s - (n/d)2^{N+k} = (n+1)s - \frac{n-r}{d}2^{N+k} = (n+1) \frac{ds - 2^{N+k}}{d} + \frac{(r+1)2^{N+k}}{d}$ (5.11)

等式 5.10 和 5.11 右边的范围为 $0 \leq x < 2^{N+k}$ 。对于一个 32 位的无符号整数 n 来说,取 $N=32$,使 k 满足 $2^k < d \leq 2^{k+1}$,然后设置 $s=(2^{N+k}+2^k)/d$ 。如果 $ds \geq 2^{N+k}$,那么 $n/d = ns \gg (N+k)$;否则, $n/d = (ns + s) \gg (N+k)$ 。另外,如果 d 是 2 的幂,还可以使用移位运算来代替除法。

【例 5.14】 `udiv_by_const` 函数测试了上述方法。

在这里, d 是一个固定的常数,而不是一个变量。可以预先计算好 s 和 k 的值,运行时只须包含与实际值 d 相关的计算。

```
unsigned int udiv_by_const ( unsigned int n, unsigned int d )
{
    unsigned int s, k, q;

    /* 假定 d != 0 */
}
```

```

/* 先找到 k 且  $(1 \ll k) \leq d < (1 \ll (k+1))$  */
for (k=0; d/2 >= (1u << k); k++);
if (d== 1u << k)
{
    /* 用移位来执行除法 */
    return n >> k;
}

/* d 的范围为  $(1 \ll k) < d < (1 \ll (k+1))$  */
s = (unsigned int) ((1ull << (32+k)) + (1ull << k)) / d;

if ((unsigned long long)s*d >= (1ull << (32+k)))
{
    /*  $n/d = (n*s) \gg (32+k)$  */
    q = (unsigned int) ((unsigned long long)n*s >> 32);
    return q >> k;
}

/*  $n/d = (n*s+s) \gg (32+k)$  */
q = (unsigned int) ((unsigned long long)n*s + s >> 32);
return q >> k;
}

```

137

如果事先知道 n 的范围为 $0 \leq n < 2^{31}$, 相当于一个正的整型数, 那么就无须操心多种不同的情况。每次 k 加 1 也不用担心 s 是否会溢出。取 $N=31$, 选择 k 满足 $2^{k-1} < d \leq 2^k$, 并且设置 $s = (2^{N+k} + 2^k - 1) / d$, 那么 $n/d = (ns) \gg (N+k)$ 。

5.10.4 除数是常数的有符号除法

可使用与 5.10.3 小节类似的想法和运算规则来解决除数是有符号常数的除法。如果 $d < 0$, 那么先进行除数是 $|d|$ 的除法, 然后再修正符号位。将 5.10.3 小节中的第一个数学结论延伸到有符号数 n , 如果 $d > 0$ 和 $2^{N+k} < ds \leq 2^{N+k} + 2^k$, 那么

$$n/d = (ns) \gg (N+k), (0 \leq n < 2^N) \quad (5.12)$$

$$n/d = (ns) \gg (N+k) + 1, (-2^N \leq n < 0) \quad (5.13)$$

对 32 位的有符号数 n , 取 $N=31$, 使 $k \leq 31$, 并满足 $2^{k-1} < d \leq 2^k$ 。这就可以确保能找到一个满足上述关系的 32 位无符号数 $s = (2^{N+k} + 2^k) / d$ 。要特别注意 32 位的有符号数 n 和

32 位的无符号数 s 相乘, 这里使用一个 signed long long 类型的乘法结合符号位修正, 来实现这个乘法运算。

【例 5.15】 显示如何实现除数是有符号数 d 的除法。

实际上, 在编译时已先计算好了 k 和 s 的值, 运行时只须进行对特定值 d 的涉及 n 的运算。

```
int sdiv_by_const(int n, int d)
{
    int s, k, q;
    unsigned int D;

    /* 设定 D 是 d 的绝对值, 假定 d != 0 */
    if (d > 0)
    {
        D = (unsigned int)d;          /* 1 ≤ D ≤ 0x7FFFFFFF */
    }
    else
    {
        D = (unsigned int)-d;         /* 1 ≤ D ≤ 0x80000000 */
    }

    /* 首先找到 k, 使得 (1 << k) ≤ D < (1 << (k+1)) */
    for (k = 0; D/2 >= (1u << k); k++);

    if (D == 1u << k)
    {
        /* 用移位来执行 */
        q = n >> 31;                 /* 0, if n > 0; -1, if n < 0 */
        q = n + ((unsigned)q >> (32 - k)); /* insert rounding */
        q = q >> k;                   /* divide */
        if (d < 0)
        {
            q = -q;                   /* correct sign */
        }
        return q;
    }

    /* Next find s in the range 0 ≤ s ≤ 0xFFFFFFFF */
```



```

/* Note that k here is one smaller than the k in the equation */
s = (int)((1ull << (31 + (k + 1))) + (1ull << (k + 1)))/D);

if (s >= 0)
{
    q = (int)(((signed long long)n * s) >> 32);
}
else
{
    /* (unsigned)s = (signed)s + (1 << 32) */
    q = n + (int)(((signed long long)n * s) >> 32);
}
q = q >> k;

/* if n < 0 then the formula requires us to add one */
q += (unsigned)n >> 31;

/* if d was negative we must correct the sign */
if (d < 0)
{
    q = -q;
}

return q;
}

```

139

7.3 节将介绍如何用汇编语言来实现高效的除法运算。

小 结 除 法

- 尽可能避免使用除法。对环形缓冲区的处理可以不用除法。
- 如果不能避免除法运算,那么尽可能考虑使用除法程序同时产生商 n/d 和余数 $n \% d$ 的好处。
- 对于重复对同一除数 d 的除法,预先计算好 $s = (2^k - 1)/d$ 。可用乘以 s 的 $2k$ 位乘法来代替除以 d 的 k 位无符号整数除法。
- 对于无符号被除数 $n < 2^N$,除数是无符号常数 d 的除法,可以找到一个 32 位的无符号数 s 和移位 k ,满足 n/d 或是 $(ns) \gg (N+k)$,或是 $(ns+s) \gg (N+k)$ 。究竟是哪一个,由 d 决定。对有符号的除法,也有类似的结果。

5.11 浮点运算

大多数 ARM 处理器硬件上并不支持浮点运算。这样在一个对价格敏感的嵌入式应用中,可节省空间和降低功耗。除了硬件向量浮点累加器 VFP 和 ARM7500FE 上的浮点累加器 FPA 外,C 编译器必须在软件上提供浮点支持。

实际上,这意味着 C 编译器要把每一个浮点操作转换成一个子程序的调用。C 库函数中的子程序使用整型运算来模拟浮点操作。这些代码是用高度优化的汇编语言编写的。尽管如此,浮点运算执行起来还是要比相应整型运算慢得多。

如果要快速执行并得到小数的值,那么就应使用定点(fixed-point)或者块浮动(block-floating)算法。在音频、视频等许多数字信号处理中,经常会用到小数/分数(fractional value)。这是软件编程的一个很大也很重要的方面,所以本书专门用一章——第 8 章,来讲述在 ARM 上的数字信号处理方法。要想获得最好的性能,就要用汇编语言来实现算法(见第 8 章的例子)

5.12 内联函数和内嵌汇编

在 5.5 节介绍了如何高效地调用函数。使用内联函数可以完全去除函数调用开销。另外许多编译器允许在 C 源程序中使用内嵌汇编。使用包含汇编的内嵌函数,可以使编译器支持通常不能有效使用的 ARM 指令和优化方法。本节的例子将在 armcc 中使用内嵌汇编。

不要把内嵌汇编程序和汇编器 armasm 或者 gas 混淆。内嵌汇编是 C 编译器的一部分。C 编译器仍然执行寄存器分配、函数进入和退出等。同时编译器也会试图优化所写的内嵌汇编,或者为调试模式(debug mode)分开优化。尽管 C 代码编译输出的结果在功能上与内嵌汇编程序是等价的,但实际上它们是不完全相同的。

内联函数和内嵌汇编最大的好处是,可以实现一些在 C 语言部分中通常难以完成的操作。使用内联函数要比使用 #define 宏定义更好,因为后者不检查函数参数和返回值的类型。

来看一个在许多语音处理算法中经常使用的运算:饱和双倍乘累加。这是对 16 位的有符号数 x 和 y 与 32 位的累加值 a 进行 $a+2xy$ 运算。另外,所有的操作如果超过 32 位的范围,则将取最接近的值(饱和运算)。 x 和 y 是 Q15 定点整数,因为它们呈现的值是 $x2^{-15}$ 和 $y2^{-15}$ 。类似地, a 是 Q31 定点整数,因为它呈现的值是 $a2^{-31}$ 。

使用内联函数 `qmac` 来实现这个新的操作：

```
__inline int qmac_v1(int a, int x, int y)
{
    int i;

    i = x * y; /* this multiplication cannot saturate */
    if (i >= 0)
    {
        /* x * y is positive */
        i = 2 * i;
        if (i < 0)
        {
            /* the doubling saturated */
            i = 0x7FFFFFFF;
        }
        if (a + i < a)
        {
            /* the addition saturated */
            return 0x7FFFFFFF;
        }
        return a + i;
    }
    /* x * y is negative so the doubling can't saturate */
    if (a + 2 * i > a)
    {
        /* the accumulate saturated */
        return -0x80000000;
    }
    return a + 2 * i;
}
```

现在可以用这个新的操作来实现一个饱和相关性(saturating correlation)了。换句话说,就是饱和计算 $a = 2x_0y_0 + \dots + 2x_{N-1}y_{N-1}$ 。

```
int sat_correlate_v1(short * x, short * y, unsigned int N)
{
    int a = 0;
```

```

do
{
    a = qmac_v1(a, *(x++), *(y++));
} while ( -- N);
return a;
}

```

编译器会用内联代码来代替每个 qmac 函数调用。换句话说,就是插入代码来代替调用 qmac。qmac 的 C 语言实现需要多条 if 语句,效率并不很高。可以用部分汇编语言写出效率更高的代码,因为 C 编译器下的内嵌汇编功能允许在内嵌的 C 函数中使用汇编。

【例 5.16】 显示使用内嵌汇编高效实现 qmac 函数。

这个例子支持 armcc 和 gcc 2 种内嵌汇编格式。它们有较大的差异。在 gcc 模式下,“cc”告知编译器指令需要读或者写条件代码标志。详细信息可参考 armcc 或 gcc 手册。

```

__inline int qmac_v2(int a, int x, int y)
{
    int i;
    const int mask = 0x80000000;

    i = x * y;
    #ifdef __ARMCC_VERSION                /* check for the armcc compiler */
        __asm
        {
            ADDS    i, i, i                /* double */
            EORVS i, mask, i, ASR # 31    /* saturate the double */
            ADDS    a, a, i                /* accumulate */
            EORVS a, mask, a, ASR # 31    /* saturate the accumulate */
        }
    #endif
    #ifdef __GNUC__                        /* check for the gcc compiler */
        asm("ADDS    %0, %1, %2          ":"=r" (i); "r" (i) , "r" (i); "cc");
        asm("EORVS %0, %1, %2, ASR # 31  ":"=r" (i); "r" (mask), "r" (i); "cc");
        asm("ADDS    %0, %1, %2          ":"=r" (a); "r" (a) , "r" (i); "cc");
        asm("EORVS %0, %1, %2, ASR # 31  ":"=r" (a); "r" (mask), "r" (a); "cc");
    #endif

    return a;
}

```

这个内嵌代码把主循环体中的指令从 19 条减少到 9 条。

【例 5.17】 假设使用带有 ARMv5E 扩展指令的 ARM9E 处理器,使用新的 ARMv5E 指令重写 qmac 函数:

```
__inline int qmac_v3(int a, int x, int y)
{
    int i;

    __asm
    {
        SMULBB i, x, y    /* multiply */
        QDADD a, a, i     /* double + saturate + accumulate + saturate */
    }
    return a;
}
```

这时,主循环体编译只用了 6 条指令:

```
sat_correlate_v3_s
    STR    r14,[r13,#-4]!    ; stack lr
    MOV    r12,#0            ;a=0
sat_v3_loop
    LDRSH  r3,[r0],#2        ;r3=*(x++)
    LDRSH  r14,[r1],#2       ;r14=*(y++)
    SUBS   r2,r2,#1          ;N-- and set flags
    SMULBB r3,r3,r14         ;r3=r3*r14
    QDADD  r12,r12,r3        ;a=sat(a+sat(2*r3))
    BNE    sat_v3_loop      ;if (N!=0) goto loop
    MOV    r0,r12            ;r0=a
    LDR    pc,[r13],#4      ;return r0
```

还有其它一些通常 C 不支持的指令,包括协处理器指令,例 5.18 说明了如何使用协处理器指令。

【例 5.18】 通过写入协处理器 15(CP15)来清空指令 cache。
可使用类似的代码来访问其它的协处理器。

```
void flush_Icache(void)
{
    #ifdef __ARMCC_VERSION /* armcc */
```

```

    __asm { MCR p15, 0, 0, c7, c5, 0 }
#endif
#ifdef __GNUC__ /* gcc */
    asm ( "MCR p15, 0, r0, c7, c5, 0" );
#endif
}

```

小结 内联函数和内嵌汇编

- 使用内联函数来声明新的操作或者 C 编译器不支持的基本操作。
- 使用内嵌汇编可以利用到 C 编译器不支持的 ARM 指令, 比如协处理器指令或者 ARMv5E 扩展指令。

5.13 移植问题

这里是把代码从其它处理器移植到 ARM 上可能会碰到的一些问题的总结。

- **char 类型** 在 ARM 上, 默认 char 是无符号的, 而不像许多其它处理器那样认为是有符号的。一个经常会碰到的问题是: 在循环体中, 使用一个 char 类型的循环计数值 i , 循环继续的条件是 $i \geq 0$, 这在 ARM 上就变成了无穷的循环。在这种情况下, armcc 会出现一个“无符号数与 0 比较”(unsigned comparison with zero) 的警告。可通过编译器选项使得 char 变成有符号的类型, 或者把循环计数值改成 int 类型, 来解决这个问题。
- **int 类型** 一些老的体系结构的 int 类型是 16 位的, 当这些 16 位的 int 类型需要移到 ARM 的 32 位 int 类型上时(尽管这种情况不多见), 可能会带来一些问题。因为表达式在求值前已经被转换成 int 类型, 因此, 如果 $i = -0x1000$, 那么表达式 $i == 0xf000$ 在 16 位的机器上是真(true), 而在 32 位的机器上却是假(false)。^{*}
- **不对齐的数据指针** 一些处理器支持从不对齐的地址装载 short 和 int 类型数据的值。C 程序可以直接操作指针, 这样可能会造成边界不对齐, 比如, 把一个 char * 转换成 int *。直至 ARMv5TE, ARM 体系结构都不支持边界不对齐的指针。为了发现这样的问题, 在 ARM 上运行的程序要使用边界检查。比如, 可以配置 ARM720T, 把边界不对齐的访问定义为数据异常。
- **字节排列方式(大端/小端)** C 代码可以对内存系统的字节排列方式进行假设, 比如, 把 char * 转换成 int *。如果在 ARM 上配置的字节排列方式与代码预期的一

* 在 16 位机器上, i 的补码是 0xf000, 而在 32 位机器上, i 的补码是 0xfffff000。——译者注

样,那么就不会有什么问题;否则,就要改写代码,使之不依赖字节排列方式。更多细节见 5.9 节。

- **函数原型** armcc 编译器传递参数是窄的(narrow),意味着数据要缩小到参数类型的范围。如果函数原型不正确,那么函数可能会返回错误的结果。其它编译器传递参数是宽的(wide),即使函数原型不正确,也可能产生正确的结果。应该总是使用 ANSI 原型。
- **位域(bit-fields)的使用** 在一个位域中,各个位的编排是与字节排列方式有关的。如果 C 代码假定是以固定的次序来编排各个位的,那么这个代码就不是可移植的。
- **枚举 enumerations)的使用** 虽然 enum 是可移植的,但不同的编译器会对一个枚举类型分配不同数目的字节。gcc 编译器分配 4 字节给一个枚举类型。如果一个枚举只有 8 位的值,则 armcc 编译器只分配 1 字节。所以,如果在一个 API 结构体内使用了枚举类型,那么就不能在不同的编译器之间对代码和库进行交叉连接。
- **内嵌汇编** 在 C 代码中使用内嵌汇编会降低不同体系结构之间代码的可移植性。可以把内嵌汇编分成几个小的容易被替换的内嵌函数,用普通 C 语言来实现这些函数,作为参考,对用在其它体系结构上是很有帮助的。
- **volatile 关键字** 对 ARM 的存储器映像(memory-mapped)外设端口类型定义要使用 volatile 关键字。这个关键字阻止编译器优化相关的存储器访问,也保证编译器生成正确类型的数据访问。例如,如果定义了一个存储器位置(外设端口)是 volatile short 类型,那么编译器将会使用 16 位的 load-store 指令 LDRSH 和 STRH 来访问它。*

145

5.14 总 结

通过一定的风格来编写 C 程序,可以帮助 C 编译器生成执行速度更快的 ARM 代码。对性能有严格要求的应用,经常会包含一些对系统性能起决定作用的关键程序,那么就要使用本章所列举的方法重点编写这些程序。

这里是本章涉及的性能关键点:

- 对局部变量、函数参数和返回值要使用 signed 和 unsigned int 类型。这样可以避免类型转换,而且可高效地使用 ARM 的 32 位数据操作指令。
- 最高效的循环体形式是减计数到零(counts down to zero)的 do-while 循环。
- 展开重要的循环来减小循环开销。

* 使用 volatile 关键字还可以避免 cache 的影响。——译者注

ARM 嵌入式系统开发

- 不要依赖编译器来优化掉重复的储存器访问。指针别名会阻止编译器的这种优化。
- 尽可能把函数参数的个数限制在 4 个以内。如果函数参数都存放在寄存器内,那么函数调用就会快得多。
- 按元素尺寸从小到大排列的方法来安排结构体,特别是在 thumb 模式下编译。
- 不要使用位域,可以用掩码和逻辑操作来替代。
- 避免除法,可以用倒数的乘法来替代。
- 避免边界不对齐的数据。如果数据有可能边界不对齐,那么就要使用 `char *` 指针类型来访问。
- 在 C 编译器中使用内嵌汇编可以利用到 C 编译器本来不支持的指令或者优化。

第 6 章

ARM 汇编与优化

- 编写汇编代码
- 性能分析和周期计数
- 指令调整
- 寄存器分配
- 条件执行
- 循环结构
- 位操作
- 高数的 switch
- 边界不对齐数据的处理
- 总 结

ARM 嵌入式系统开发

在一个嵌入式软件系统中,经常包含一些决定整个系统性能的关键程序,通过优化这些程序,可以降低系统的功耗和实时操作所需的时钟频率。优化可以把一个不可行的系统变成可行,也可以把一个毫无竞争力的系统变得极有竞争力。

如果认真地使用第 5 章中介绍的方法来编写 C 代码,那么就会获得相对较高的程序执行效率。但若要想获得最好的性能,就要通过手写汇编来优化那些关键程序。手工编写汇编代码,可以直接控制在 C 语言编程时不能有效使用的 3 个优化工具:

- **指令调整*** 调整一段代码中的指令序列,以避免处理器的暂停等待。ARM 指令执行是在指令流水线中进行的,所以一条指令执行的时间会受其相邻指令的影响。在 6.3 节中将着重讨论这个问题。
- **寄存器分配** 决定如何分配变量给 ARM 寄存器或者堆栈,以获得最好的性能。目标是要使访问存储器的次数降到最少,详见 6.4 节。
- **条件执行** 可以使用 ARM 条件代码和条件指令的全部功能,详见 6.5 节。

优化汇编程序需要付出很多额外的努力,因而不必费力去优化那些对性能影响不大的程序。花时间去优化一个程序,其实也会有一些额外的收获,比如对运算法则、程序瓶颈及数据流等问题都会有更好的理解。

6.1 节将首先介绍 ARM 上的汇编程序编程方法,说明了如何把一个 C 函数替换为一个可以优化的汇编函数。

然后描述了针对 ARM 汇编的一般优化技术。这里不再专门论述 Thumb 汇编,因为使用 32 位数据总线时,32 位的 ARM 汇编会有更好的性能。Thumb 对减小 C 代码编译后的目标代码大小是最有帮助的,而在 16 位数据总线上,对性能和程序执行效率没有太大影响。当然,这里涉及的许多方法对 ARM 和 Thumb 是同样有效的。

对一个程序最好的优化是应根据目标硬件板上 ARM 核的不同而改变优化方法,特别是针对数字信号处理(在第 8 章将详细讨论)。当然,也可以写出一个对所有 ARM 核都相当有效的代码。为了保持一致,本章所有例子都使用针对 ARM9TDMI 的优化和周期计数。这些例子也可以有效地运行在从 ARM7TDMI 到 ARM10E 的所有 ARM 核上。

6.1 编写汇编代码

本节给出的例子说明了如何编写基本的汇编代码。这里,假定读者熟悉第 3 章描述的 ARM 指令集(完整的指令集参见附录 A),也熟悉 5.4 节中描述的 ARM 和 Thumb 过程调

* 这里的指令调整是指由人工或编译器在代码执行之前所做的静态指令流调整,与代码执行时由处理器完成的指令动态调度是不同的。——译者注

用标准(ATPCS)。

与本书的其它部分一样,本章所有例子都使用 ARM 宏汇编器 `armasm`(关于 `armasm` 语法和参考见附录 A.4 节),也可以使用 GNU 汇编器 `gas`(关于 GNU 汇编语法见 A.5 节)。

【例 6.1】 描述通常汇编优化的第一步:如何把一个 C 函数转换为汇编函数。

看下面一个简单的 C 程序 `main.c`:打印整数 0~9 的平方数。

```
# include <stdio.h>
int square(int i);

int main(void)
{
    int i;

    for (i=0;i<10;i++)
    {
        printf("Square of %d is %d\n", i, square(i));
    }
}

int square(int i)
{
    return i*i;
}
```

来看一下如何将 `square` 函数改写成执行结果相同的汇编函数。去除 `square` 中除声明(第 2 行)以外的其它 C 代码,创建一个新的 C 文件 `main1.c`,然后添加 `armasm` 汇编文件 `square.s`。汇编文件内容如下:

```
AREA    |.text|, CODE, READONLY

EXPORT  square

;int square(int i)
square
    MUL    r1, r0, r0        ;r1 = r0 × r0
    MOV    r0, r1            ;r0 = r1
    MOV    pc, lr            ;return r0
```

END

保留字 AREA 命名了代码所在的区域或段(section)。如果使用了非阿拉伯数字的字符作为标号或名字,那么最好用 2 条垂直的线把名字括起来;否则,许多非阿拉伯数字的字符就会表示其它一些特殊的含义。在上面的代码中,定义了一个名为 .text 的只读代码区。

保留字 EXPORT 表示符号 square 可以用作外部连接。在第 6 行定义了符号 square 作为一个代码标号。

注意: armasm 把不缩进的正文作为一个标号定义。

square 被调用时,参数传递由 ATPCS(见 5.4 节)定义。输入参数通过寄存器 r0 传递,最后返回值也通过寄存器 r0 返回。ARM 乘法指令有一个限制,就是目标寄存器不能和第一个参数寄存器相同,所以先把乘法结果放到 r1,然后再送到 r0。

保留字 END 表示汇编文件的结尾,分号表示注释的开始。

下面的命令举例说明了如何用命令行工具来生成这个例子的可执行目标文件。

```
armcc -c main1.c
armasm squsre.s
armlink -o main1.axf main1.o square.o
```

例 6.1 必须用 ARM 代码方式(32 位)来编译,才能正确执行。如果以 Thumb 代码方式(16 位)来编译,那么汇编子程序必须使用 BX 指令来返回。

【例 6.2】 当从以 Thumb 方式编译的 C 程序中调用 ARM 汇编代码时,例 6.1 的汇编代码惟一要改变的是把返回指令改用 BX。BX 指令会根据 lr 寄存器的位 0 来判断是返回到 ARM,还是返回到 Thumb 状态。因此这个子程序既可以在 ARM 状态下,也可以在 Thumb 状态下被调用。只要处理器支持 BX 指令(ARMv4T 及其以上),就可用 BX lr 来代替 MOV pc, lr。创建一个新的汇编文件 square2.s 如下:

```
AREA    |.text|, CODE, READONLY

EXPORT  square

;int square(int i)
square
    MUL    r1, r0, r0      ;r1 = r0 × r0
    MOV    r0, r1          ;r0 = r1
    BX     lr              ;return r0
```

END

使用 Thumb C 编译器 `tcc` 来构建这个例子。设置交叉工作方式允许,这样连接器就可以允许 Thumb C 代码调用 ARM 汇编代码。使用下面的命令来重建这个例子:

```
tcc -c main1.c
armasm -apcs/interwork square2.s
armalink -o main2.axf main1.o square2.o
```

【例 6.3】 说明如何在一个汇编程序里调用一个子程序。

把例 6.1 整个程序(包括 `main`)转换成汇编程序,然后调用 C 库函数中的 `printf` 子程序。创建一个新的汇编文件 `main3.s`,内容如下:

```
AREA    |.text|, CODE, READONLY

EXPORT  main

IMPORT  |Lib$$Request $$armlib|, WEAK
IMPORT  __main      ;C library entry
IMPORT  printf      ;prints to stdout

i       RN 4

        ,int main(void)
main
        STMFD    sp!, {i, lr}
        MOV      i, #0
loop
        ADR      r0, print_string
        MOV      r1, i
        MUL      r2, i, i
        BL       printf
        ADD      i, i, #1
        CMP      i, #10
        BLT      loop
        LDMFD    sp!, {i, pc}

print_string
        DCB      "Square of %d is %d\n", 0
```

END

这里,使用了一个新的保留字 IMPORT,以声明其它文件中定义的标号。引入的标号 Lib \$\$Request \$\$armlib 表示要求连接器连接一个标准的 ARM C 库。WEAK 表示如果这个标号在连接时找不到,那么制止连接器给出一个错误。如果标号找不到,那么它的值将会是 0。引入标号 __main 是 C 库初始化代码的开始。如果定义了自己的 main,那么就只需要引入这些标号,在 C 代码中定义的 main 将自动引入这些标号。引入 printf 将允许程序去调用 C 库函数。

保留字 RN 表示允许使用寄存器的名字(不是寄存器编号)。在这里,定义 i 作为寄存器 r4 的替换名字。使用寄存器名字将使代码更具可读性,而且对以后改变变量在寄存器中的分配也会容易得多。

在 ATPCS 中,函数必须保护寄存器 r4~r11 和 sp。由于会改变 i(r4),并在调用 printf 时会改变 lr 的值,因此,在函数的开始,使用 STMFD 指令来把这 2 个寄存器的值压入堆栈;LDMFD 指令再从堆栈恢复这些寄存器,并且通过把返回地址写入 pc 来返回。

保留字 DCB 用来定义一个字符串,或者以逗号分隔的多个字节数据。

使用下面的命令行脚本来构建这个例子:

```
armasm main3.s
armlink - o main3.axf main3.o
```

注意例 6.3 也假定代码从 ARM 状态被调用。如果代码要像例 6.2 那样可以从 Thumb 状态下被调用,那么必须能够返回到 Thumb 状态。对于 ARMv5 以前的体系结构,必须使用 BX 指令来返回。可以把最后一条指令替换为下面 2 条指令:

```
LDMFD SP!,{i,LR}
BX     LR
```

最后,来看一个例子,这个例子传递了多于 4 个的函数参数。ATPCS 把前 4 个参数放在寄存器 r0~r3。其余的参数存放在堆栈里。

【例 6.4】 定义一个函数 sumof,可以求任意数目的整数的和。

参数是求和整数的个数和一串求和的整数。函数 sumof 用汇编语言编写,可以接受任意多个数目的参数。把这个例子的 C 代码部分放在文件 main4.c:

```
#include <stdio.h>

/* N 是列表中求和整数的个数... */
int sumof(int N, ...);
```

```

int main(void)
{
    printf("Empty sum = %d\n", sumof(0));
    printf("1 = %d\n", sumof(1,1));
    printf("1 + 2 = %d\n", sumof(2,1,2));
    printf("1 + 2 + 3 = %d\n", sumof(3,1,2,3));
    printf("1 + 2 + 3 + 4 = %d\n", sumof(4,1,2,3,4));
    printf("1 + 2 + 3 + 4 + 5 = %d\n", sumof(5,1,2,3,4,5));
    printf("1 + 2 + 3 + 4 + 5 + 6 = %d\n", sumof(6,1,2,3,4,5,6));
}

```

下一步在汇编文件 `sumof.s` 中定义 `sumof` 函数：

```

        AREA      |.text|, CODE, READONLY

        EXPORT    sumof

;int sumof(int N, ...)
sumof
    SUBS        N, N, #1        ;do we have one element
    MOVLT       sum, #0         ;no elements to sum!
    SUBS        N, N, #1        ;do we have two elements
    ADDGE       sum, sum, r2
    SUBS        N, N, #1        ;do we have three elements
    ADDGE       sum, sum, r3
    MOV         r2, sp          ;top of stack
loop
    SUBS        N, N, #1        ;do we have another element
    LDMGEFD     r2!, {r3}       ;load from the stack
    ADDGE       sum, sum, r3
    BGE         loop
    MOV         r0, sum
    MOV         pc, lr          ;return r0

    END

```

ARM 嵌入式系统开发

代码把需要和整数的剩余数目计数保存在 N。前 3 个值在寄存器 r1, r2 及 r3 中, 剩下的值放在堆栈。可以用下面的命令行来构建这个例子:

```
armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o
```

6.2 性能分析和周期计数

任何优化的第一步都是要找出对性能影响较大的程序段, 并且测量它们目前的性能状况。CPU 性能分析器(profiler)可用来测量在每一个子程序上所花的时间比例和执行周期。因此, 可以使用 CPU 性能分析器来判断对性能影响最大的程序。周期计数器(cycle counter)用来测量一个特定程序段所占用的周期数。使用周期计数器来衡量某个子程序优化前后的性能情况, 可以评估优化工作的成效。

ADS1.1 调试器使用的 ARM 软件仿真器(simulator)叫做 ARMulator, 提供了性能分析和周期计数的功能。ARMulator 性能分析器以一定的时间间隔来对程序计数器 pc 进行采样。性能分析器标识 pc 所指向的函数, 并更新与之相遇的每一个函数的命中计数器的值。另外还可以使用性能分析器跟踪的输出作为一个源文件来进一步分析。

必须了解所使用的性能分析器是如何工作的, 以及它的精度限制。如果基于 pc 采样的性能分析器记录的采样点太少, 那么产生的结果也就毫无意义。也可以采用硬件系统的 pc 采样性能分析器, 即利用定时器中断来采集 pc 数据。

注意: 定时器中断会降低正在被测量的系统的运行速度!

ARM 提供的工具通常并不包含周期计数硬件部件。最简单的做法还是使用 ARM 调试器所带的 ARM 软件仿真器来测量周期计数。对于不同的硬件平台, 可以把 ARMulator 配置为仿真不同的 ARM 核来获得周期计数基准。

6.3 指令调整*

指令的执行时间依赖于流水线的实现。本章内容假定使用 ARM9TDMI 的流水线定

* 这里的指令调整(instruction scheduling)是指在程序运行前, 由编译器或人工完成的静态调度。与计算机体系结构中讲述的指令动态调度(由处理器完成)是不同的。——译者注

时,在附录 D 的 D.3 节可以找到相应的内容。下面的规则总结了 ARM9TDMI 上一些公共指令的周期数情况。

指令执行依赖于在 cpsr 中的条件标志。如果条件不匹配,那么指令占用一个周期;如果条件匹配,那么就依照下面的规则:

- ALU 操作,比如加法、减法和逻辑操作,占用一个周期,包括由一个立即数决定的移位。如果使用特定的寄存器移位,那么就要增加一个周期。如果指令是写入 pc 的,那么要增加 2 个周期。
- 从存储器装载 N 个 32 位字的装载指令,比如 LDR 和 LDM,将占用 N 个周期。但是要注意,接下来的一个周期,装入的最后一个字还不能被使用。^{*}再下一个周期是更新装载地址。这里假设是针对没有 cache 的零等待的存储器,或是带有 cache 并 cache 命中的情况。只装载一个值的 LDM 指令是个例外,需要花费 2 个周期。如果指令装载 pc,那么也要增加 2 个周期。
- 从存储器装载 16 位或者 8 位数据的装载指令,如 LDRB, LDRSB, LDRH 和 LDRSH,占用 1 个周期。在接下来的 2 个周期中,装入的结果还不能被使用。再下一个周期是更新装载地址。同样,这里假设是没有 cache 的零等待的存储器,或者是带有 cache 并 cache 命中的情况。
- 分支(Branch)指令占用 3 个周期。
- 存储指令中,存储 N 个值占用 N 个指令周期。这里假设是没有 cache 的零等待的存储器,或者是 cache 命中,或带有 N 个自由入口的写缓冲器的 cache 系统。如果是只存储一个值的 STM 指令,要花费 2 个周期。
- 乘法指令依赖于第二个操作数的值,会有多个不同周期数(见 D.3 节中的表 D.6)。

为了理解在 ARM 上如何有效地进行指令调整,就需要了解 ARM 流水线及其相关性(dependencies)。ARM9TDMI 处理器可并行地执行 5 个操作:

- 取指(Fetch) 在地址 pc 处从存储器中取出指令。指令被装载到内核中,然后进入指令流水线。
- 译码(Decode) 对前一个周期中取到的指令进行译码。如果操作数还没准备好,那么处理器可以通过前向通道之一从寄存器堆中读入操作数。
- ALU 执行前一个周期译码的指令。注意这条指令是从地址 pc-8(ARM 状态)或者 pc-4(Thumb 状态)取到的。通常,这一步包括了计算数据操作的结果,或计算装载、存储、跳转操作的地址。在这一步,一些指令会花费几个周期。例如,乘法和寄存器控制的移位操作会占用几个 ALU 周期。

^{*} 要延时一个周期才可以使用最后一个装载的数据,详见附录 D。——译者注

ARM 嵌入式系统开发

- *LS1* 通过装载/存储指令来装载/存储特定的数据。如果不是装载或者存储指令,那么这个步骤没有任何作用。
- *LS2* 对通过字节或者半字装载指令装载的数据进行截取(extract)和左端补0(zero-),或符号位扩展(sign-extend)。如果指令不是装载一个8位字节或16位半字的,那么该步骤也没有任何作用。

图 6.1 是 5 级 ARM9TDMI 流水线的简化功能图。

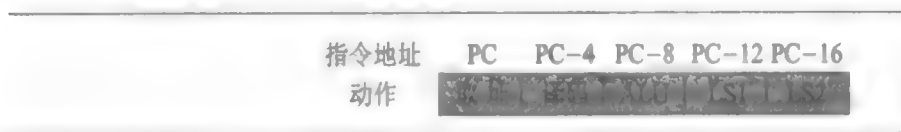


图 6.1 ARM9TDMI 在 ARM 状态时的流水线执行情况

注意：该图并没有表示出乘法和寄存器移位操作的情况。

一条指令完成了流水线的 5 个步骤后,最后结果写入寄存器。

注意：pc 指向的是正在取的那条指令的地址。ALU 正在执行的指令是前面从 pc-8 地址取到的,同时从当前 pc 地址取新的指令。

流水线是如何影响指令执行时间的呢? 考虑下面的例子。这些例子说明了指令执行的周期数是如何改变的——由于当前指令在流水线中继续之前,必须等前一条指令完成某个步骤;并且使用附录 D 总结的不同 ARM 核的指令周期时间和互锁(interlock)周期来计算一段代码需要花费的周期数。

如果一条指令需要前一条指令的执行结果,而这时结果还没有出来,那么处理器就会等待。这称为流水线相关(pipeline hazard)或者流水线互锁(pipeline interlock)。

【例 6.5】 没有互锁的情况。

```
ADD    r0, r0, r1
ADD    r0, r0, r2
```

这两条指令占用了 2 个周期。ALU 用 1 个周期计算 $r0+r1$,因此这个结果在第 2 个周期 ALU 计算 $r0+r2$ 时已经准备好了。

【例 6.6】 显示由于装载产生了 1 个周期互锁。

```
LDR    r1,[r2, #4]
ADD    r0,r0, r1
```

这 2 条指令占用了 3 个指令周期。在第 1 个周期,ALU 计算地址 $r2+4$,同时译码 ADD 指令。但是,因为装载指令还没有把 r1 的值装载进来,ADD 在第 2 个周期就不能继

续下去。因此流水线等待了 1 个周期,直到装载指令完成 LS1 步骤。当 r1 准备好后,处理器在第 3 个周期 ALU 执行 ADD 操作。

图 6.2 举例说明了互锁是如何影响流水线的。ADD 指令在流水线的 ALU 阶段暂停了 1 个周期,等待装载指令完成 LS1 步骤。图中这个暂停用斜体的 ADD 表示。由于 LDR 指令在流水线继续执行,而 ADD 指令中止了,它们之间就出现了一个间隙。这个间隙有时被称作流水线气泡(pipeline bubble),这里用破折号来表示这个气泡。

流水线	取指	译码	ALU	LS1	LS2
周期 1	...	ADD	LDR	...	
周期 2		...	<i>ADD</i>	LDR	...
周期 3		...	ADD	—	LDR

图 6.2 由于装载引起的 1 个周期的互锁

【例 6.7】 显示由于延迟装载引起的 1 个周期的互锁。

```
LDRB    r1,[r2,#1]
ADD     r0,r0,r2
EOR     r0,r0,r1
```

这 3 条指令占用了 4 个周期。虽然 ADD 紧跟在装载字节后的周期进行,但是 EOR 指令不能在第 3 个周期开始。寄存器 r1 的值一直要到装载指令完成流水线的 LS2 阶段才会准备好。处理器中止 EOR 指令一个周期。

注意 ADD 指令一点也没有影响流水线时序。无论 ADD 指令是否存在,整个过程总是占用 4 个指令周期! 图 6.3 显示了这个指令序列是如何通过处理器流水线的。由于 ADD 指令不使用 r1 这个装载的结果,所以 ADD 指令不会引起任何等待。

流水线	取指	译码	ALU	LS1	LS2
周期 1	EOR	ADD	LDRB	...	
周期 2	...	EOR	ADD	LDR	...
周期 3		...	<i>EOR</i>	ADD	LDR
周期 4		...	EOR	—	ADD

图 6.3 由于延迟装载引起的 1 个周期的互锁

【例 6.8】 说明为什么分支指令要占用 3 个周期。处理器在跳转到一个新的地址时必须刷新流水线。

```
MOV     r1,#1
```

ARM 嵌入式系统开发

```

B      case1
AND    r0, r0, r1
EOR    r2, r2, r3
...

case1
SUB    r0, r0, r1

```

3 条执行的指令一共占用了 5 个周期。MOV 指令在第 1 个周期执行。在第 2 个周期，分支指令计算目标地址。这时需要刷新流水线，用新的 pc 值来填入。重新填入花费了 2 个周期。最后，SUB 指令正常执行。图 6.4 显示了流水线每一个周期的状态。流水线在跳转发生时，丢弃了分支指令之后的 2 条指令。

流水线	取指	译码	ALU	LS1	LS2
周期 1	AND	B	MOV	...	
周期 2	EOR	AND	B	MOV	...
周期 3	SUB	—	—	B	MOV
周期 4	...	SUB	—	—	B
周期 5		...	SUB	—	

图 6.4 由于跳转引起的流水线刷新

load 指令的调整

装载指令在编译后的代码中出现的频度很高，计算下来大约占有所有指令的 1/3。所以，仔细安排装载指令的时间次序，可以防止流水线中止，改善性能。编译器会尽量安排好代码执行的时间顺序，但是 5.6 节中描述的指针别名问题会影响编译器优化。如果不能保证 load 和 store 指令中的 2 个指针不指向同一个地址，那么编译器就不能去除在 store 指令之前的 load 指令。

来看一个关于密集存储器访问(memory-intensive)任务的例子。下面的函数 str_tolower 把一串以 0 结束的字符串从 in 复制到 out，并在处理过程中把字符串转换成小写字母。

```

void str_tolower(char * out, char * in)
{
    unsigned int c;

    do
    {

```

```

    c = *(int ++);
    if(c >= 'A' && c <= 'Z')
    {
        c = c + ('a' - 'A');
    }
    *(out++) = (char)c;
}while(c);
}

```

ADS1.1 编译器生成了下面的输出结果。

注意：编译器优化了条件 $(c \geq 'A' \ \&\& \ c \leq 'Z')$ ，变为检查 $0 \leq c - 'A' \leq 'Z' - 'A'$ ，因为编译器可以使用一条无符号比较指令来完成这个检查。

```

str_tolower
    LDRB    r2, [r1], #1        ; t = *(int++)
    SUB     r3, r2, #0x41        ; r3 = c - 'A'
    CMP     r3, #0x19           ; if(c <= 'Z' - 'A')
    ADDLS   r2, r2, #0x20        ; c += 'a' - 'A'
    STRB    r2, [r0], #1        ; *(out++) = (char)c
    CMP     r2, #0              ; if(c != 0)
    BNE     str_tolower         ; goto str_tolower
    MOV     pc, r14             ; return

```

遗憾的是，在 LDRB 指令装载了 c 之后，SUB 指令立即使用 c 的值，这样 ARM9TDMI 流水线将等待 2 个周期。由于 load 指令之后的每一条指令都要用到 c 这个值，因此编译器不能做任何更好的改进。但是，有两种方法，可以通过使用汇编，改变算法结构来消除等待周期。这里把这些方法称为通过预载 (preloading) 和循环展开 (unrolling) 来调整、安排 load 指令。

1. 通过预载来调整 load 指令

这种安排装载时序的方法，就是在本次循环的最后装载下次循环所需要的数据，而不是在本次循环的开始装载数据。为了不增加代码量，将不对循环进行展开。

【例 6.9】 str_tolower 使用了预载的方法。

```

out    RN 0    ; pointer to output string
in     RN 1    ; pointer to input string
c      RN 2    ; character loaded
t      RN 3    ; scratch register

```

```

;void str_tolower_preload(char * out, char * in)
str_tolower_preload
LDRB    c, [in], #1        ;c = *(in++)
loop
SUB      t, c, #'A'        ;t = c - 'A'
CMP      t, #'Z' - 'A'     ;if (t <= 'Z' - 'A')
ADDLS    c, c, #'a' - 'A'   ; c += 'a' - 'A';
STRB     c, [out], #1      ;*(out++) = (char)c;
TEQ      c, #0             ;test if c == 0
LDRNEB   c, [in], #1       ;if (c != 0) { c = *in++;
BNE      loop              ; goto loop;}
MOV      pc, lr            ;return

```

这个程序比原来的 C 版本多了一条指令,但是在循环体内部少了 2 个指令周期。这使得在 ARM9TDMI 上每一个字符的处理循环从 11 个周期减少到 9 个周期,性能提高了 1.22 倍。

因为 ARM 指令是可以条件执行的,所以 ARM 体系结构特别适合这种类型的预载。由于循环 i 装载的数据是为循环 $i+1$ 准备的,这样对于第一次和最后一次循环就有问题了。对第一次循环,可以在循环开始前通过增加额外的 load 指令来预载数据;但对最后一次循环,循环体已不需要再读取任何数据,读取的可能是超过数组边界的数据,这可能会引起数据异常中止!对于 ARM,可以通过 load 指令的条件执行来解决这个问题。在例 6.9 中,只有在下一次循环将继续的情况下,才会预载下一个字符。这样,最后一次循环就不会装载任何数据。

2. 通过循环展开来调整 load 指令

这种调整 load 指令的方法是通过循环展开来插入复制循环体的内容。例如,一次循环执行原来 $i, i+1$ 及 $i+2$ 的循环体的内容,当循环 i 中的某个操作结果还没有准备好时,可以先执行循环 $i+1$ 中的操作,以避免等待循环 i 的结果。

【例 6.10】对 str_tolower 函数使用了循环展开的方法来调整 load 指令。

```

out      RN 0      ;pointer to output string
in       RN 1      ;pointer to input string
ca0      RN 2      ;character 0
t        RN 3      ;scratch register
ca1      RN 12     ;character 1
ca2      RN 14     ;character 2

```

```

;void str_tolower_unrolled(char *out, char *in)
str_tolower_unrolled
    STMFD    sp!, {lr}                ;function entry
loop_next3
    LDRB     ca0, [in], #1             ;ca0 = *in++;
    LDRB     ca1, [in], #1             ;ca1 = *in++;
    LDRB     ca2, [in], #1             ;ca2 = *in++;
    SUB      t, ca0, #'A'               ;convert ca0 to lower case
    CMP      t, #'Z'-'A'
    ADDLS    ca0, ca0, #'a'-'A'
    SUB      t, ca1, #'A'               ;convert ca1 to lower case
    CMP      t, #'Z'-'A'
    ADDLS    ca1, ca1, #'a'-'A'
    SUB      t, ca2, #'A'               ;convert ca2 to lower case
    CMP      t, #'Z'-'A'
    ADDLS    ca2, ca2, #'a'-'A'
    STRB     ca0, [out], #1            ;*out++=ca0;
    TEQ      ca0, #0                    ;if (ca0!=0)
    STRNEB   ca1, [out], #1            ;*out++=ca1;
    TEQNE    ca1, #0                    ;if (ca0!=0 && ca1!=0)
    STRNEB   ca2, [out], #1            ;*out++=ca2;
    TEQNE    ca2, #0                    ;if (ca0!=0 && ca1!=0 && ca2!=0)
    BNE      loop_next3                ;goto loop_next3;
    LDMFD    sp!, {pc}                ;return;

```

可以看到,这个循环的执行效率是最高的。在 ARM9TDMI 上对每个字符的处理操作只需要 7 个周期,比最初的 `str_tolower` 性能提高了 1.57 倍。而且由于 ARM 指令能条件执行,避免了访问超过字符串结尾的字符。

但是,例 6.10 的改进也带来了一些额外的开销:代码量比原来增加了 2 倍。上面的代码假定在输入字符串的结尾后可以再读取 2 个字符,但如果字符串正好处在 RAM 的最后,那么就会产生数据异常。另外,对于很短的字符串,执行效率可能也会下降,这是因为:① 堆栈操作 `lr` 会引起额外的函数调用开销;② 在发现最后 2 个字符是越界字符前,程序可能已毫无意义地对这 2 个字符进行了操作。

在确定数据量比较大的情况下,对实时要求高的部分应用程序使用循环展开的方法会比较适合。而且,如果在编译时已经知道了数据量的大小,那么就可以解决越过数组边界的读取问题。

小结 指令调整

- ARM 核是流水线结构的。如果指令的执行结果是后续指令的源操作数,那么处理器将会插入等待周期直至数据准备好,这样流水线就会产生几个周期的延迟。
- load 和乘法指令在许多情况下会产生延迟。对特定的 ARM 处理器核,关于指令的周期定时和延迟可参见附录 D。
- 有 2 种软件方法可以解决由 load 指令导致的流水线互锁:预载——在循环 i 中预载循环 i+1 的数据;循环展开——在一次循环中插入原循环体 i 和 i+1 的代码。

6.4 寄存器分配

可以使用 ARM 16 个可见寄存器中的 14 个来保存通用数据,另外 2 个是堆栈指针 r13 和程序计数器 r15。对于一个遵循 ATPCS 调用规则的函数,必须保护被调用寄存器 r4~r11 的值。ATPCS 同时也规定了堆栈应是 8 字节边界对齐的,因此在调用子程序时必须保护这个边界。对优化过的、需要很多寄存器的汇编程序,可以使用下面的模板来保护寄存器:

```
routine_name
    STMFd    sp!, {r4-r12, lr}      ;stack saved registers
    ;body of routine
    ;the fourteen registers r0-r12 and lr are available
    LDMFD    sp!, {r4-r12, pc}      ;restore registers and return
```

栈操作(压栈和退栈)r12 的惟一目的是,保证堆栈是 8 字节边界对齐的。如果程序不调用其它的 ATPCS 子程序,那么无需对 r12 进行栈操作。对于 ARMv5 及其以上的体系结构,甚至当程序从 Thumb 代码中被调用时,也可以使用上面的模板。但在 ARMv4T 处理器上,如果程序可能从 Thumb 代码中被调用,那么应该把模板修改为:

```
routine_name
    STMFd    sp!, {r4-r12, lr}      ;stack saved registers
    ;body of routine
    ;registers r0-r12 and lr are available
    LDMFD    sp!, {r4-r12, lr}      ;restore registers
    BX       lr                     ;return, with mode switch
```

下面我们将着重于分析对寄存器需求比较大的任务该如何较好地 把变量分配到寄存器,如何处理局部变量超过 14 个的情况,以及如何最好地利用 14 个可用的寄存器。

6.4.1 分配变量给寄存器

编写一个汇编程序时,最好为变量使用寄存器名字,而不是直接使用寄存器编号(如 r5)。这样可以容易地更改变量分配的寄存器编号。如果变量不会交迭使用,那么还可以对同一个物理寄存器使用不同的寄存器名字。使用寄存器名字可以提高代码的清晰度和可读性。

对寄存器来说,大部分的 ARM 操作都是正交的。换句话说,特定的寄存器并没有特定的角色。在一个程序里,如果把 2 个寄存器 Ra 和 Rb 在所有出现的地方都互换,那么程序的功能并不会改变。但是,也有一些情况寄存器的物理编号是重要的:

- **参数寄存器** ATPCS 规范定义了一个函数的前 4 个参数是分配在寄存器 r0~r3 的。其它的参数存放在堆栈,返回值必须存放在 r0。
- **一个多次装载或存储操作所使用的寄存器** 一条多寄存器装载或存储的 LDM 和 STM 指令,总是操作一组编号按增序排列的寄存器。如果 r0 和 r1 出现在操作寄存器中,那么处理器将总是在低地址装载或存储 r0,然后是 r1,等等。
- **装载和存储双字** 在 ARMv5E 上引入的 LDRD 和 STRD 指令,总是使用一对连续编号的寄存器——Rd 和 Rd+1,而且,Rd 必须是偶数编号寄存器(如 r0,r2,r4 等)。

举例来看编写汇编程序时如何分配寄存器。假定想要对存储器中 N 位数的一个数组向上移 k 位。为了简单起见,假定 N 是一个很大的数,而且是 256 的倍数,同时假定 $0 \leq k < 32$,输入/输出指针都是字边界对齐的。这种乘以 2^k 类型的操作,对于处理多精度数的算术运算是较普遍的。而且对于从一种位或字节边界对齐,到另一种位或字节边界对齐的块拷贝操作是很有用的。例如,C 库函数 memcopy 可以只用字访问来复制一字节的数组。

C 函数 shift_bits 实现了一个简单的对 N 位数的 k 位移位操作:

```
unsigned int shift_bits(unsigned int * out, unsigned int * in, unsigned int N, unsigned int k)
{
    unsigned int carry = 0, x;
    do
    {
        x = * in++;
        * out++ = (x << k) | carry;
        carry = x >> (32 - k);
        N -= 32;
    } while(N);
    return carry;
}
```

ARM 嵌入式系统开发

提高效率最明显的方法是,展开循环来一次处理 8 个字 256 位的数据。这样就可以利用多寄存器装载或存储的 load 和 store 指令,一次装载和存储 8 个字的数据,以获得最高效率。先不考虑寄存器数量,编写出下面的代码:

```

shift_bits
    STMFD    sp!, {r4-r11, lr}        ;save registers
    RSB      kr, k, #32                ;kr = 32 - k;
    MOV      carry, #0

loop
    LDMIA    in!, {x_0-x_7}            ;load 8 words
    ORR      y_0, carry, x_0, LSL k    ;shift the 8 words
    MOV      carry, x_0, LSR kr        ;recall x_0 = y_1
    ORR      y_1, carry, x_1, LSL k
    MOV      carry, x_1, LSR kr
    ORR      y_2, carry, x_2, LSL k
    MOV      carry, x_2, LSR kr
    ORR      y_3, carry, x_3, LSL k
    MOV      carry, x_3, LSR kr
    ORR      y_4, carry, x_4, LSL k
    MOV      carry, x_4, LSR kr
    ORR      y_5, carry, x_5, LSL k
    MOV      carry, x_5, LSR kr
    ORR      y_6, carry, x_6, LSL k
    MOV      carry, x_6, LSR kr
    ORR      y_7, carry, x_7, LSL k
    MOV      carry, x_7, LSR kr
    STMIA    out!, {y_0-y_7}          ;store 8 words
    SUBS     N, N, #256                ;N -= (8 words * 32 bits)
    BNE      loop                     ;if (N!= 0) goto loop;
    MOV      r0, carry                 ;return carry;
    LDMFD    sp!, {r4-r11, pc}

```

现在来看寄存器分配。输入参数并不需要移动寄存器,可以立即指派:

```

out    RN 0
in     RN 1
N      RN 2
k      RN 3

```

要使多次装载能正确工作,必须把 $x_0 \sim x_7$ 分配给编号连续递增的寄存器, $y_0 \sim y_7$ 也一样。注意在开始 y_1 操作之前,已完成了对 x_0 的操作,所以可以对 x_n 和 y_{n-1} 指派同一个编号的寄存器。指派如下:

x_0	RN 5
x_1	RN 6
x_2	RN 7
x_3	RN 8
x_4	RN 9
x_5	RN 10
x_6	RN 11
x_7	RN 12
y_0	RN 4
y_1	RN x_0
y_2	RN x_1
y_3	RN x_2
y_4	RN x_3
y_5	RN x_4
y_6	RN x_5
y_7	RN x_6

但是还有一个问题,剩下 2 个变量 $carry$ 和 kr ,但只有一个空闲的寄存器 lr 。有几种可能的方法来处理这个问题(寄存器不够分配):

- 在每一个循环中减少操作,以减少所需要的寄存器数目。这种情况下,load 可以每次只装载 4 个字的数据,而不是 8 个字。
- 使用堆栈来存储最少使用的值,以释放一些寄存器。这种情况下,可以把循环计数值 N 存放在堆栈(关于交换寄存器给堆栈的更多详情见 6.4.2 小节)。
- 改变代码的实现,以释放更多的寄存器。下面我们就使用这种解决方法(更多例子见 6.4.3 小节)。

我们经常重申,一个算法的实现过程要通过多次调整寄存器分配,直至算法适合于 14 个有效寄存器。在上述例子中可以发现,变量 $carry$ 根本不需要一直放在同一个寄存器中!开始时, $carry$ 先放在 y_0 ,当 x_0 不再需要时,转移到 y_1 ,依次类推。这样就可以把 kr 分配给 lr 来完成这个程序,因为 $carry$ 不再需要新的寄存器分配。

【例 6.11】 最后优化过的 `shift_bits` 程序,使用了全部 14 个可用的 ARM 寄存器。

kr RN lr

```

shift_bits
    STMFD    sp!, {r4 - r11, lr}    ;save registers
    RSB      kr, k, #32              ;kr = 32 - k;
    MOV      y_0, #0                 ;initial carry

loop
    LDMIA    in!, {x_0 - x_7}        ;load 8 words
    ORR      y_0, y_0, x_0, LSL k    ;shift the 8 words
    MOV      y_1, x_0, LSR kr        ;recall x_0 = y_1
    ORR      y_1, y_1, x_1, LSL k
    MOV      y_2, x_1, LSR kr
    ORR      y_2, y_2, x_2, LSL k
    MOV      y_3, x_2, LSR kr
    ORR      y_3, y_3, x_3, LSL k
    MOV      y_4, x_3, LSR kr
    ORR      y_4, y_4, x_4, LSL k
    MOV      y_5, x_4, LSR kr
    ORR      y_5, y_5, x_5, LSL k
    MOV      y_6, x_5, LSR kr
    ORR      y_6, y_6, x_6, LSL k
    MOV      y_7, x_6, LSR kr
    ORR      y_7, y_7, x_7, LSL k
    STMIA    out!, {y_0 - y_7}      ;store 8 words
    MOV      y_0, x_7, LSR kr
    SUBS     N, N, #256              ;N -= (8 words * 32 bits)
    BNE      loop                   ;if (N!= 0) goto loop;
    MOV      r0, y_0                 ;return carry;
    LDMFD    sp!, {r4 - r11, pc}

```

6.4.2 使用超过 14 个的局部变量

如果需要在程序中使用多于 14 个 32 位的局部变量,那么就必须把一些变量存放在堆栈中。基本的做法是,从一个算法的最内层循环向外考察,因为最内层循环对性能的影响最大。

【例 6.12】显示 3 层嵌套循环,每一层循环需要从它的外层循环继承的状态信息(关于循环构造的更多方法和例子见 6.6 节)。

```

nested_loops
    STMFD    sp!,    {r4 - r11,lr}
    ;set up loop 1
loop1
    STMFD    sp!,    {loop1 registers}
    ;set up loop 2
loop2
    STMFD    sp!,    {loop2 registers}
    ;set up loop 3
loop3
    ;body of loop 3
B{cond} loop3
    LDMFD    sp!,    {loop2 register}
    ;body of loop 2
B{cond} loop2
    LDMFD    sp!,    {loop1 register}
    ;body of loop 1
B{cond} loop1
    LDMFD    sp!,    {r4 - r11,pc}

```

可以发现,即使使用例 6.12 中的构造,对最内层循环还是没有足够的寄存器,那就需要交换内部循环变量到外部堆栈。如果直接用数字作为堆栈地址的偏移量,那么对于汇编代码来说是很难维护和调试的,汇编器会在分配变量到堆栈时自动计算偏移量。

【例 6.13】 显示如何使用 ARM 汇编保留字 MAP(别名~)和 FIELD(别名#),以在堆栈中为变量和数组定义和分配空间。这 2 个保留字与 C 中的 struct 操作有相似的作用。

	MAP	0	;map symbols to offsets starting at offset 0
a	FIELD	4	;a is 4 byte integer (at offset 0)
b	FIELD	2	;b is 2 byte integer (at offset 4)
c	FIELD	2	;c is 2 byte integer (at offset 6)
d	FIELD	64	;d is an array of 64 characters (at offset 8)
length	FIELD	0	;length records the current offset reached

example

```

STMFD    sp!, {r4 - r11, lr}    ;save callee registers
SUB      sp, sp, #length        ;create stack frame
;...
STR      r0, [sp, #a]           ;a = r0;

```

```

LDRSH    r1, [sp, # b]      ;r1 = b;
ADD      r2, sp, # d        ;r2 = &d[0]
;...
ADD      sp, sp, # length   ;restore the stack pointer
LDMFD    sp!, {r4 - r11, pc} ;return

```

6.4.3 最大限度地使用寄存器

在像 ARM 这样的 load-store 体系结构处理器上,访问寄存器中的数据要比访问存储器中的数据效率高很多。这里有一些关于把几个小于 32 位(sub-32-bit)长度的变量存放在一个 32 位寄存器的窍门,这样做可以减小代码尺寸,并改善性能。本小节列举了 3 个例子,以说明如何把多个变量打包(pack)在一个 ARM 寄存器。

【例 6.14】 假定需要通过一个可编程的增量来遍历数组。一个普通的例子是通过一个可变采样率的声音来步进产生不同音调的音符。用 C 代码表示如下:

```

sample = table[index]
index += increment

```

通常变量 index 和 increment 都是很小的,足以保存在一个 16 位变量中。把这 2 个变量打包放入一个 32 位的变量 indinc:

$$\text{indinc} = (\text{index} \ll 16) + \text{increment} =$$

bit31	16	15	0
index		increment	

把这行 C 代码转换成汇编,使用一个寄存器来存放 indinc:

```

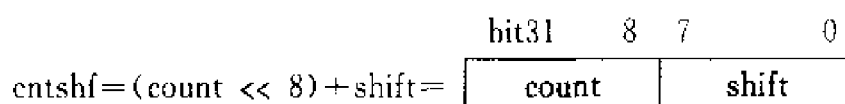
LDRB    sample, [table, indinc, LSR# 16] ;table[index]
ADD     indinc, indinc, indinc, LSL# 16   ;index += increment

```

注意: 如果 index 和 increment 都是 16 位的值,那么把 index 放进 indinc 的高 16 位也正确实现了 16 位环绕(16-bit-wrap-around),即 $\text{index} = (\text{short})(\text{index} + \text{increment})$ 。这种做法对循环缓冲区(circular buffer)的操作也是很有帮助的。

【例 6.15】 如果使用寄存器中的数值作为移位次数,ARM 使用寄存器的位 0~7 作为移位值,并忽略寄存器的位 8~31,那么就可以用位 8~31 来存放另一个变量。

这个例子显示了如何组合寄存器指定的移位值 shift 和循环计数值 count,把 40 个元素的数组右移 shift 位。我们定义一个新的变量 cntshf 来存放 count 和 shift:



```

out      RN 0      ;输出数组地址
in       RN 1      ;输入数组地址
cntshf   RN 2      ;计数和移位值
x        RN 3      ;临时变量

```

```
void shift_right(int * out, int * in, unsigned shift);
```

```
shift_right
```

```
    ADD      cntshf, cntshf, # 39 << 8 ;count = 39
```

```
shift_loop
```

```
    LDR      x, [in], # 4
```

```
    SUBS     cntshf, cntshf, # 1 << 8 ;decrement count
```

```
    MOV      x, x, ASR cntshf ;shift by shift
```

```
    STR      x, [out], # 4
```

```
    BGE      shift_loop ;continue if count ≥ 0
```

```
    MOV      pc, lr
```

【例 6.16】 在处理 8 位或 16 位值的数组时,有时可以把多个变量同时放入一个 32 位的寄存器中。这称为单发射多数据(Single Issue Multiple Data, SIMD)处理。

直至 ARMv5 的体系结构版本并不显式地支持 SIMD 操作。但是,仍然有些地方可以进行 SIMD 处理。6.6 节将介绍如何把多个循环变量存放在一个寄存器里。这里将研究一个关于图像的例子,使用普通的 ADD 和 MUL 指令处理一幅图像中的多个 8 位像素,以完成一些 SIMD 操作。

假定需要把两幅图像 X 和 Y 合并成一幅新的图像 Z。分别用 x_n , y_n 和 z_n 来表示这些图像中的第 n 个 8 位像素。设范围为 0~256 的 a 为比例因子(scaling factor)。为合并图像,设

$$z_n = (ax_n + (256 - a)y_n)/256 \quad (6.1)$$

换句话说,图像 Z 是图像 X 以比例 $a/256$ 加到以 $1 - (a/256)$ 为比例的图像 Y 上。

注意 $z_n = w_n / 256$

这里 $w_n = a(x_n - y_n) + 256y_n \quad (6.2)$

因此每个像素需要一条减法、一条乘法、一条移位加法和一条右移指令。为了一次处理多个像素,使用一条字装载指令一次装载 4 个像素。也就是把 4 个变量打包在一个字中:

ARM 嵌入式系统开发

$$[x_3, x_2, x_1, x_0] = x_3 2^{24} + x_2 2^{16} + x_1 2^8 + x_0 =$$

Bit 24	16	8	0
x_3	x_2	x_1	x_0

然后把 8 位的数据通过一条 AND 指令和屏蔽寄存器将其变成 16 位数据。使用下面的符号：

$$[x_2, x_0] = x_2 2^{16} + x_0 =$$

Bit 31	16	15	0
x_2	x_0		

注意：如果使用数学等式 $a2^{16} + b$ 来表示 $[a, b]$ ，那么符号数 $[a, b] + [c, d] = [a + b, c + d]$ 。因此可以使用普通的算术指令来实现 SIMD 操作。

下面的代码显示了如何使用 2 个乘法来一次处理 4 个像素。这些代码假定处理 176×144 大小的 1/4GIF 图像。

```

IMAGE_WIDTH    EQU 176 ,QCIF width
IMAGE_HEIGHT    EQU 144 ,QCIF height

pz      RN 0      ;pointer to destination image (word aligned)
px      RN 1      ;pointer to first source image (word aligned)
py      RN 2      ;pointer to second source image (word aligned)
a       RN 3      ;8-bit scaling factor (0-256)

xx      RN 4      ;holds four x pixels [x3, x2, x1, x0]
yy      RN 5      ;holds four y pixels [y3, y2, y1, y0]
x       RN 6      ;holds two expanded x pixels [x2, x0]
y       RN 7      ;holds two expanded y pixels [y2, y0]
z       RN 8      ;holds four z pixels [z3, z2, z1, z0]
count   RN 12     ;number of pixels remaining
mask    RN 14     ;constant mask with value 0x00ff00ff

;void merge_images(char *pz, char *px, char *py, int a)
merge_images
    STMFD    sp!, {r4-r8, lr}
    MOV      count, # IMAGE_WIDTH * IMAGE_HEIGHT
    LDR      mask, = 0x00FF00FF    ;[    0, 0xFF,    0, 0xFF]

merge_loop
    LDR      xx, [px], #4          ;[  x3,  x2,  x1,  x0]
```



```

LDR    yy, [py], #4      ;[  y3,  y2,  y1,  y0]
AND    x, mask, xx       ;[  0,  x2,  0,  x0]
AND    y, mask, yy       ;[  0,  y2,  0,  y0]
SUB    x, x, y           ;[  (x2-y2),  (x0-y0)]
MUL    x, a, x           ;[  a*(x2-y2),  a*(x0-y0)]
ADD    x, x, y, LSL#8    ;[          w2,          w0]
AND    z, mask, x, LSR#8 ;[  0,  z2,  0,  z0]
AND    x, mask, xx, LSR#8 ;[  0,  x3,  0,  x1]
AND    y, mask, yy, LSR#8 ;[  0,  y3,  0,  y1]
SUB    x, x, y           ;[  (x3-y3),  (x1-y1)]
MUL    x, a, x           ;[  a*(x3-y3),  a*(x1-y1)]
ADD    x, x, y, LSL#8    ;[          w3,          w1]
AND    x, mask, x, LSR#8 ;[  0,  z3,  0,  z1]
ORR    z, z, x, LSL#8    ;[  z3,  z2,  z1,  z0]
STR    z, [pz], #4      ;store four z pixels
SUBS   count, count, #4
BGT    merge_loop
LDMFD  sp!, {r4-r8, pc}

```

由于

$$0 \leq w_n \leq 256a + 255(256 - a) = 256 \times 255 = 0xFF00 \quad (6.3)$$

以上代码是有效的。

因此,可以很容易地通过分别处理数据的高 16 位和低 16 位,把值[w2,w0]分成 w2 和 w0。我们已成功地通过 32 位装载、存储和数据操作指令并行地处理了 4 个 8 位像素。

小结 寄存器分配

- ARM 有 14 个通用寄存器:r0~r12 和 r14。堆栈指针寄存器 r13 和程序计数器 r15 不能用于通用数据。操作系统中断时,经常假定用户模式下的 r13 指向一个有效的堆栈,因此不要试图去使用 r13。
- 如果使用的局部变量超过了 14 个,那么从最内层循环向外把剩余的变量放入堆栈。
- 在编写汇编程序时,应尽量使用寄存器名字,而不是物理寄存器编号。这样做可便于重新分配寄存器和维护代码。
- 为了有效利用寄存器,有时可以把多个变量存放在一个寄存器中。比如,把循环计数值和移位值放在一个寄存器中,也可以把多个像素存放在一个寄存器中。

6.5 条件执行

ARM 指令集的一个重要特征就是大多数的指令均可包含一个可选的条件码。当程序状态寄存器 cpsr 中的条件标志满足指定条件时,带条件码的指令才会执行。条件执行基于 15 个条件码之一。若没有指定条件,则汇编器默认为无条件执行(AL),其它 14 个条件被分成 7 对。这些条件依赖于 cpsr 寄存器中的 4 个条件标志 N,Z,C,V。各种 ARM 条件码见附录 A 的表 A.2。关于条件执行的介绍也可参见 2.2.6 小节和 3.8 节。

默认情况下,ARM 指令并不会更新 ARM 寄存器 cpsr 中的 N,Z,C,V 标志。对大多数的指令,若要更新这些标志,则需要对指令助记符加后缀 S。例外的是不写入目标寄存器的比较指令,它们惟一的目的是更新这些标志,因此不需要 S 后缀。

通过组合使用条件执行和条件标志设置,可以简单地实现 if 语句,而不需要任何分支指令。这样可以改善性能,因为分支指令会占用较多的周期数;同时也可以减小代码尺寸。

【例 6.17】 下面的 C 代码把一个 unsigned integer 类型的 $i(0 \leq i \leq 15)$ 转换成一个十六进制的字符 c:

```
if (i < 10)
{
    c = i + '0';
}
else
{
    c = i + 'A' - 10;
}
```

用汇编语言的条件执行来重写这个例子。如下:

```
CMP     i, #10
ADDLO   c, i, # '0'
ADDHS   c, i, # 'A' - 10
```

第 1 条 ADD 指令没有改变条件标志。第 2 条 ADD 指令也是通过比较的结果有条件地执行。6.3.1 小节显示了一个类似的例子,有条件地转换成小写字母。

条件执行对于多重条件结构会更加有效。

【例 6.18】 下面的 C 代码表示了字符 c 是元音字母时的情况

```

if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' )
{
    vowel++;
}

```

利用条件执行编写的汇编代码如下：

```

TEQ    c, # 'a'
TEQNE  c, # 'e'
TEQNE  c, # 'i'
TEQNE  c, # 'o'
TEQNE  c, # 'u'
ADDEQ  vowel, vowel, #1

```

一旦 TEQ 比较指令执行的结果匹配, cpsr 寄存器中的条件标志 Z 就会被设置为 0。这样在条件 Z=0 的情况下, 下面的 TEQNE 指令就不会被执行。类似地, 对 vowel 增量的 ADDEQ 指令也有同样的效果。这样对于 if 语句中的同类型比较, 都可以使用这种方法。

【例 6.19】 看下面的 C 代码, 判别字符 c 是否是字母:

```

if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}

```

为了能够高效地执行这个函数, 使用加法或者减法指令来判断字符 c 的范围; 再用无符号比较来判断字符是否在这个范围之内。下面的汇编程序高效地实现了这个函数:

```

SUB     temp, c, # 'A'
CMP     temp, # 'Z' - 'A'
SUBHI   temp, c, # 'a'
CMPHI   temp, # 'z' - 'a'
ADDLS   letter, letter, #1

```

包括 switch 语句的更多复杂的情况见 6.8 节。

注意: 表 6.1 给出了逻辑操作 AND 和 OR 之间的转换关系。可以把一个 OR 的逻辑表达式转换成 AND 的逻辑表达式, 这对于简化或重新排列逻辑表达式是有帮助的。

表 6.1 逻辑变换关系

反转表达式	等价表达式
$\neg(a \& b)$	$(\neg a) \mid (\neg b)$
$\neg(a \mid b)$	$(\neg a) \& (\neg b)$

小结 条件执行

- 利用条件执行可实现大部分 if 条件语句,比使用条件分支指令效率高得多;
- 可使用带有条件码的比较指令来实现带有几个类似的逻辑 AND 或者 OR 的 if 条件语句。

6.6 循环结构

大部分对性能影响较大的程序都会包含循环体。在 5.3 节中提到使用减计数到零 (count down to zero) 结构的 ARM 循环执行速度是最快的。本节将会讨论如何用汇编来实现高效的循环体;同时也会介绍如何展开循环,以获得最好的性能。

6.6.1 减计数循环

对一个 N 次的减计数循环来说,循环计数值 i 计数为 N~1,循环的结束条件是 i=0。一个较好的实现是:

```
MOV i,N
loop
;loop body goes here and i = N,N-1,...,1
SUBS i,i, #1
BGT loop
```

这个循环开销包括一条设置条件标志的减法指令和一条条件分支指令。在 ARM7 和 ARM9 上,一次循环占用 4 个指令周期。如果 i 是数组的下标值,那么 i 的范围为 N-1~0,从而可以访问下标值为 0 的数组元素。实现的代码如下:

```
SUBS i,N, #1
loop
;loop body goes here and i = N-1,N-2,...,0
SUBS i,i, #1
BGE loop
```

只有最后一次循环才置位标志 Z,在循环的其它过程中,标志 Z 保持为 0。在循环结束时,也可以使用条件 EQ 和 NE 来满足一些其它的需要。比如,如果要为下一个循环预载数据(在 6.3 小节讨论过这个问题),那么就要避免在执行最后一次循环时预载数据,可使用条件 NE 来完成预载操作。

每一次循环计数值并不一定都是减 1。如果需要 $N/3$ 次循环(假设 N 是 3 的倍数),那么计数值每次减 3,循环体的执行效率就会更高。

```
MOV    i,N
loop
    ;loop body goes here and iterates ( round up ) (N/3) times
    SUBS    i,i,#3
    BGT     loop
```

6.6.2 展开计数循环

又回到了循环展开的主题。通过多次执行循环内部语句来展开循环,这样可以减小循环开销,但也会带来一些问题。如果循环计数值不是展开次数的倍数怎么办?如果循环计数值比展开次数小怎么办?在 5.3 节中也讨论过类似的问题。本小节将讨论如何用汇编语言来解决这些问题。

我们把 C 库函数 `memset` 作为研究的例子。这个函数把从地址 s 开始的 N 个字节的存储器单元赋值为 c 。为了提高这个函数的执行效率,将着眼于对输入操作数不增加额外限制的情况下,分析如何展开循环。`memset` 函数原型如下:

```
void my_memset ( char * s,int c, unsigned int N );
```

要提高效率,就要使用 `STR` 或者 `STM` 指令一次写入多个字节。因此,首先需要将数组指针 s 边界对齐。但是,只有在 N 足够大的情况下,才值得这样做。在不能确定“足够大的”准确值之前,先假定一个极限值 T_1 ,在 $N \geq T_1$ 的前提下,数组指针 s 边界对齐。而且 $T_1 \geq 3$,因为如果数据大小不到 4 字节,那么就没有字边界对齐的问题了。

现在假定数组 s 边界对齐,使用批量存储指令来对存储单元赋值。例如,一次循环使用 4 条批量存储指令,每条指令写 8 个字,这样每次循环就可以操作 128 字节(32 个字)的存储单元。但是,只有在 $N \geq T_2 \geq 128$ 的前提下才值得这么做,这里 T_2 是另一个极限值。

最后,还剩下 $N < T_2$ 个字节要设置。可使用 `STR` 指令一次存储 4 字节,直到 $N < 4$,再使用 `STRB` 指令单字节操作剩余的字节单元。

【例 6.20】 显示循环展开的 `memset` 函数。

根据前面提到的方法,把程序分成 3 段。由于没有找到最佳的 T_1 和 T_2 值,程序还须进一步讨论。

```
s      RN 0      ;current string pointer
c      RN 1      ;the character to fill with
N      RN 2      ;the number of bytes to fill
```

```

c_1    RN 3    ;copies of c
c_2    RN 4
c_3    RN 5
c_4    RN 6
c_5    RN 7
c_6    RN 8
c_7    RN 12

```

```

;void my_memset(char * s, unsigned int c, unsigned int N)

```

```

my_memset

```

```

;First section aligns the array

```

```

CMP     N, #T_1          ;We know that T_1 ≥ 3
BOC     memset_1ByteBlk  ;if (N < T_1) goto memset_1ByteBlk
ANDS    c_1, s, #3        ;find the byte alignment of s
BEQ     aligned          ;branch if already aligned
RSB     c_1, c_1, #4       ;number of bytes until alignment
SUB     N, N, c_1         ;number of bytes after alignment
CMP     c_1, #2
STRB    c, [s], #1
STRGEB  c, [s], #1        ;if (c_1 ≥ 2) then output Byte
STRGTB  c, [s], #1        ;if (c_1 ≥ 3) then output Byte
aligned
;the s array is now aligned
ORR     c, c, c, LSL#8     ;duplicate the character
ORR     c, c, c, LSL#16    ;to fill all four bytes of c

```

```

;Second section writes blocks of 128 bytes

```

```

CMP     N, #T_2          ;We know that T_2 ≥ 128
BOC     memset_4ByteBlk  ;if (N < T_2) goto memset_4ByteBlk
STMFD   sp!, {c_2 - c_6} ;stack scratch registers
MOV     c_1, c
MOV     c_2, c
MOV     c_3, c
MOV     c_4, c
MOV     c_5, c
MOV     c_6, c
MOV     c_7, c

```

```

        SUB     N, N, #128                ;bytes left after next block
loop128 ;write 32 words = 128 bytes
        STMIA   s!, {c, c_1-c_6, c_7}    ;write 8 words
        STMIA   s!, {c, c_1-c_6, c_7}    ;write 8 words
        STMIA   s!, {c, c_1-c_6, c_7}    ;write 8 words
        STMIA   s!, {c, c_1-c_6, c_7}    ;write 8 words
        SUBS    N, N, #128                ;bytes left after next block
        BGE     loop128
        ADD     N, N, #128                ;number of bytes left
        LDMFD   sp!, {c_2-c_6}           ;restore corrupted registers
;-----
;Third section deals with left over bytes
memset_4ByteBlk
        SUBS    N, N, #4                  ;try doing 4 bytes
loop4   ;write 4 bytes
        STRGE   c, [s], #4
        SUBGES   N, N, #4
        BGE     loop4
        ADD     N, N, #4                  ;number of bytes left
memset_1ByteBlk
        SUBS    N, N, #1
loop1   ;write 1 byte
        STRGEB  c, [s], #1
        SUBGES   N, N, #1
        BGE     loop1
        MOV     pc, lr                    ;finished so return

```

现在来讨论 T_1 和 T_2 的最佳值。首先须分析不同范围的循环计数值 N 。由于算法操作是以 128 字节、4 字节和 1 字节的块大小方式进行的,所以先按这些块尺寸来分解 N :

$$N = 128N_h + 4N_m + N_l, (0 \leq N_m < 32, 0 \leq N_l < 4)$$

这样就可以分成 3 种情况来讨论。下面关于指令周期计数的详细内容,需要参考附录 D 关于指令周期定时的有关内容。

- $0 \leq N < T_1$ 在 ARM9TDMI 上,上述程序包括返回共占用 $5N+6$ 个周期。
- $T_1 \leq N < T_2$ 如果数组 s 是字对齐的,则第 1 个算法块占用了 6 个周期;否则为 10 个周期。假如每一种边界都近似相同,那么平均占用 $(6+10+10+10)/4=9$ 个周期。第 2 个算法块占用了 6 个周期。最后一块占用了 $5(32N_h+N_m)+5(N_l+Z_l)$

+2 个周期,这里如果 $N_i=0$,则 Z_i 是 1;否则 Z_i 为 0。因此这种情况下总共占用了 $5(32N_h+N_m+N_i+Z_i)+17$ 个周期。

- $N \geq T_2$ 与前一种情况相同,第 1 个算法块平均占用了 9 个周期。第 2 个算法块占用了 $36N_h+21$ 个周期。最后一个算法块占用了 $5(N_m+Z_m+N_i+Z_i)+2$ 个周期,这里如果 $N_m=0$,则 Z_i 是 1;否则 Z_i 为 0。因此这种情况下总共占用了 $36N_h+5(N_m+Z_m+N_i+Z_i)+32$ 个周期。

表 6.2 总结了这些结果。比较表中的 3 行,可以清楚地看到,只要 $N_m \geq 1$,第 2 行的值就小于第 1 行,除非 $N_m=1$ 或 $N_m=0$ 。设置 $T_1=5$,从第 1 行和第 2 行来选择最佳的周期数。只要当 $N_h \geq 1$ 时,第 3 行的值就小于第 2 行的值。因此取 $T_2=128$ 。

表 6.2 不同范围 N 值的指令周期数

N 范围	指令周期数
$0 \leq N < T_1$	$640N_h+20N_m+5N_i+6$
$T_1 \leq N < T_2$	$160N_h+5N_m+5N_i+17+5Z_i$
$N \geq T_2$	$36N_h+5N_m+5N_i+32+5Z_i+5Z_m$

这个详细的例子显示了如何使用极限值来展开比较重要的循环,以及在可能的计数输入范围内获得较好的性能。

6.6.3 多层嵌套循环

多层嵌套循环需要多少个循环计数器呢?实际上,一个计数器就可以满足对循环计数总值不超过 32 位的循环体。我们可以把多个循环计数值组合放在一个寄存器,最内层的循环计数值放在寄存器的最高位。本小节给出了一个例子,以说明如何做这件事,循环计数从 $\text{max}-1$ 减到 0,计数值变成负数时循环结束。

【例 6.21】 显示如何把 3 个循环计数值合并成一个循环计数值。

假定要把矩阵 B 与矩阵 C 相乘产生的结果放入矩阵 A ,矩阵 A, B, C 的尺寸不变;假定 R, S, T 是相对比较大但小于 256 的。

```
Matrix A;      R rows * T columns
Matrix B;      R rows * S columns
Matrix C;      S rows * T columns
```

用矩阵名字的小写字母来表示指向矩阵的指针,就是指向一个以行排列的字数组。比如,元素在 i 行、 j 列,那么 $A[i, j]$ 就是在字节地址

$$\&A[i, j] = a + 4 * (i * T + j)$$

使用 3 个循环计数值 i, j, k , 下面的 C 代码实现了这个矩阵乘法的功能:

```
#define R 40
#define S 40
#define T 40

void ref_matrix_mul ( int *a, int *b, int *c)
{
    unsigned int i, j, k;

    int sum;
    for ( i = 0; j < R; i++)
    {
        for(j = 0; j < T; j++)
        {
            /* calculate a[i, j] */
            sum = 0;
            for(k = 0; k < S; k++)
            {
                /* add b[i, k] * c[k, j] */
                sum += b[i * S + k] * c[k * T + j];
            }
            a[i * T + j] = sum;
        }
    }
}
```

这里, 有许多方法可改进程序的执行效率, 例如, 可以通过改进地址下标计算, 浓缩循环结构。给 3 个循环计数值 i, j, k 分配一个寄存器计数值 count:

Bit	31	24	23	16	15	8	7	0
count =	0	$S-1-k$	$T-1-j$	$R-1-i$				

注意: 由于 k 的原因, $S-1-k$ 的计数是从 $S-1 \sim 0$, 而不是从 $0 \sim S-1$ 。

下面的汇编代码使用一个循环计数寄存器 count 实现了矩阵乘法:

```
R    EQU 40
S    EQU 40
```

T EQU 40

a RN 0 ;points to an R rows x T columns matrix

b RN 1 ;points to an R rows x S columns matrix

c RN 2 ;points to an S rows x T columns matrix

sum RN 3

bval RN 4

cval RN 12

count RN 14

;void matrix_mul(int *a, int *b, int *c)

matrix_mul

STMFD sp!, {r4, lr}

MOV count, #(R-1) ;i = 0

loop_i

ADD count, count, #(T-1) << 8 ;j = 0

loop_j

ADD count, count, #(S-1) << 16 ;k = 0

MOV sum, #0

loop_k

LDR bval, [b], #4 ;bval = B[i,k], b = &B[i,k+1]

LDR cval, [c], #4 * T ;cval = C[k,j], c = &C[k+1,j]

SUBS count, count, #1 << 16 ;k++

MLA sum, bval, cval, sum ;sum += bval * cval

BPL loop_k ;branch if k <= S-1

STR sum, [a], #4 ;A[i,j] = sum, a = &A[i,j+1]

SUB c, c, #4 * S * T ;c = &C[0,j]

ADD c, c, #4 ;c = &C[0,j+1]

ADDS count, count, #(1 << 16) - (1 << 8) ;zero (S-1-k), j++

SUBPL b, b, #4 * S ;b = &B[i,0]

BPL loop_j ;branch if j <= T-1

SUB c, c, #4 * T ;c = &C[0,0]

ADDS count, count, #(1 << 8) - 1 ;zero (T-1-j), i++

BPL loop_i ;branch if i <= R-1

LDMFD sp!, {r4, pc}

上面的程序结构比一般的实现节约了 2 个寄存器。首先,对 count 的位 16~23 进行减 1 运算,直到结果为负数,这个做法完成了 $S-1 \sim 0$ 的循环 k 。一旦这个结果为负数,代码便用加 2^{16} 来清除位 16~31。然后通过对 count 的位 8~15 减 2^8 来完成循环 j 。可以使用一条 ARM 指令来高效地编码常数 $2^{16} - 2^8 = 0xFF00$ 。count 的位 8~15 的计数为 $T-1 \sim 0$ 。当组合的加和减的结果为负时,也就完成了循环 j 。对循环 i 也采用相同的处理方法。可以看到,ARM 对加、减法指令的宽范围常数移位处理能力,使这种处理机制非常有效。

6.6.4 其它计数循环

在循环体中,有时循环计数值是作为一个输入值参与运算的;而且,也并不是在所有情况下,循环计数值计数都是从 N 减到 1 或者从 $N-1$ 减到 0。比如,若依次从一个数据寄存器选择某些位,则需要一个每次循环乘以 2 (即 2 的幂)的屏蔽。

下面将介绍一些不同计数形式的循环结构。这些例子都只使用了一条有分支功能的条件执行指令来实现循环。

6.6.4.1 负数索引

这个循环的计数值计数 $-N \sim 0$ (包含或不包含),每次累加的大小是 STEP。

```
RSB      i,N,#0      ;i = -N
loop
    ; loop body goes here and i = -N, -N+STEP, ...,
    ADDS  i,i,#STEP
    BLT   loop        ;use BLT or BLE to exclude 0 or not
```

6.6.4.2 对数索引

这个循环结构每次以 2 的幂从 2^N 递减到 1。比如, $N=4$,计数值为 16,8,4,2,1。

```
MOV      i,#1
MOV      i,i,LSL N
loop
    ;loop body
    MOVS  i,i,LSR #1
    BNE   loop
```

下面的循环结构计数值从 N 位屏蔽到 1 位屏蔽。比如, $N=4$,计数值为 15,7,3,1。

```
MOV      i,#1
RSB      i,i,i,LSL N,      ;i = (1 << N) - 1
```

```

loop
    ;loop body (循环体)
    MOVS    i,i,LSR#1
    BNE     loop

```

小结 循环结构

- 对 ARM 来说,需要 2 条指令来实现一个计数循环:一条设置标志的减法指令和一条条件分支指令。
- 展开循环可以改善性能。但不要过度展开,因为这会影响 cache 的性能。只有对大的循环次数的循环进行展开才有意义,对于循环次数很少的循环展开并不能提高效率。
- 多层嵌套的循环只需要一个循环计数寄存器。这样做可以节省寄存器,以作它用。
- ARM 可以高效地实现以负数和对数方式索引的循环。

6.7 位操作

压缩的文件格式以位的粒度来打包数据项,以获得最高的数据密度。这些数据项或者是固定宽度的,或者是可变长度的。比如用来表示长度的域或者表示版本的域就是固定宽度的;而 Huffman 编码就是可变宽度的,因为 Huffman 编码是与特征位有关的编码方法,因而它赋予出现频度较高的信号以较短的编码,而对很少出现的信号赋予较长的编码。

本节将讨论如何高效地处理位流(bitstream)。首先论述固定宽度的编码,然后是可变宽度的编码。7.6 节会介绍一般的位操作程序,如位反转和字节排列方式(大小端)等。

6.7.1 固定宽度的位域打包和解包

如果事先已经设定了屏蔽码,那么从 ARM 寄存器的任意位置截取一个无符号的位域只需要 1 个周期;否则需要 2 个周期。截取一个带符号的位域一般需要 2 个周期,除非位域正好位于一个字的高位顶部(此时位域的最高符号位就是寄存器的最高符号位)。在 ARM 上,一般使用逻辑操作和桶形移位器来打包和解开代码,请参见下面的例子。

【例 6.22】 显示如何把寄存器 r0 中的位 4~15 提取出来,结果放到寄存器 r1。

```

;unsigned unpack with mask set up in advance
;mask = 0x0000FFF
AND    r1,mask, r0, LSR#4

```

```

; unsigned unpack with no mask
MOV    r1,r0,LSL#16      ;discard bits 16~31
MOV    r1,r1,LSR#20      ;discard bits 0~3 and zero extend

;signed unpack
MOV    r1,r0,LSL#16      ;discard bits 16~31
MOV    r1,r1,ASR#20      ;discard bits 0~3 and sign extend

```

【例 6.23】 把 r1 的值放入寄存器 r0 的特定位置中。

如果 r1 的值已经被限定在正确的范围内, r0 的相应位也被清除, 那么就只需要 1 个周期。在这个例子中, 把 r1 的 12 位数据插入到 r0 的位 4~15。

```

;pack r1 into r0
ORR    r0,r0,r1,LSL#4

```

否则就需要设置一个屏蔽寄存器:

```

;pack r1 into r0
;mask = 0x00000FFF set up in advance
AND    r1,r1,mask        ;限制 r1 范围
BIC    r0,r0,mask,LSL#4   ;清除目标位
ORR    r0,r0,r1,LSL#4     ;拼接新数据

```

6.7.2 可变宽度编码的位流打包

这里的任务是要把一系列可变长度的代码打包生成一个位流。典型的就是把长度不定的 Huffman 编码或者其它算术编码数据压缩成一个数据流。但这里并不关心所用的编码方式的效率。

首先需要仔细地了解打包数据的字节排列方式。许多压缩文件格式使用大端方式打包, 这种方式下, 第一个数据的编码放置在第一个字节的最高端。为了统一起见, 下面的例子都采用了大端方式打包。这种方式有时也被称为网络次序(network order)。图 6.5 显示了如何使用大端方式把长度不定的位编码打包成一个字节流。high 和 low 表示字节的最高位和最低位。

为了在 ARM 上高效地实现打包工作, 我们使用一个 32 位的寄存器作为缓冲器, 以大端方式来保存 4 字节的数据。换句话说, 就是把字节流中的字节 0 放在寄存器的最高 8 位。然后, 依次从高位到低位, 每次把一个编码插入寄存器。

一旦寄存器满, 就把 32 位数据存放到存储器。注意对一个大端方式的存储系统, 可以

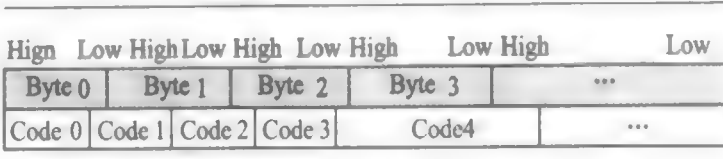


图 6.5 大端方式下位编码打包成一个字节流

不加任何修改来存储一个字;而对小端方式的存储系统,就需要在存储一个字之前调整字节顺序。

存放插入编码的寄存器称为 bitbuffer,现在还需要一个寄存器 bitsfree 来记录 bitbuffer 中没有被使用的位的数量。换句话说,bitbuffer 中包含了“32-bitsfree”个编码位,和“bitsfree”个 0 位,见图 6.6。要把 k 位的编码插入 bitbuffer,就要从 bitsfree 减去 k,然后对编码左移 bitsfree 位后插入。

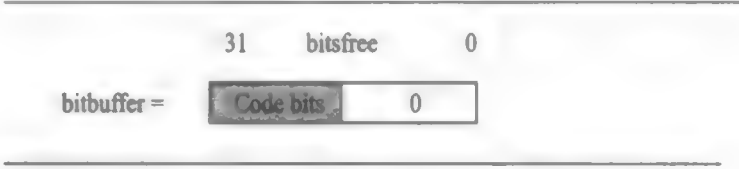


图 6.6 bitbuffer 的格式

同时也要注意边界对齐问题。由于字节流不保证字边界对齐,所以不能使用字访问来写入。为了能够采用字方式访问,可以退后到从上一个字对齐的地址开始。用 32 位的寄存器 bitbuffer 填充前面的那些数据,这样就可以使用 32 位的字进行读/写了。

【例 6.24】 提供 3 个函数 bitstream_write_start, bitstream_write_code, bitstream_write_flush。

由于这 3 个函数假定了寄存器在 2 个函数调用之间是受保护的,因此它们并不是 ATPCS 标准兼容的函数。实际上,这也不是什么问题,因为可以使用内联这些函数的代码,同时可以提高效率。

bitstream_write_start 函数的功能是使位流指针 bitstream 的边界对齐,并初始化 32 位缓冲器 bitbuffer。每次调用 bitstream_write_code 函数插入一个位长度 codebits 的值 code。最后,bitstream_write_flush 函数把剩余的字节写入位流来结束。

- bitstream RN 0 ,current byte address in the output bitstream
- code RN 4 ,current code
- codebits RN 5 ,length in bits of current code
- bitbuffer RN 6 ,32-bit output big-endian bit buffer
- bitsfree RN 7 ,number of bits free in the bit buffer
- tmp RN 8 ,scratch register

```
mask      RN 12 ;endian reversal mask 0xFFFF00FF
```

```
bitstream_write_start
```

```
    MOV     bitbuffer, #0
```

```
    MOV     bitsfree, #32
```

```
align_loop
```

```
    TST     bitstream, #3
```

```
    LDRNEB  code, [bitstream, #-1]!
```

```
    SUBNE   bitsfree, bitsfree, #8
```

```
    ORRNE   bitbuffer, code, bitbuffer, ROR #8
```

```
    BNE     align_loop
```

```
    MOV     bitbuffer, bitbuffer, ROR #8
```

```
    MOV     pc, lr
```

```
bitstream_write_code
```

```
    SUBS    bitsfree, bitsfree, codebits
```

```
    BLE     full_buffer
```

```
    ORR     bitbuffer, bitbuffer, code, LSL bitsfree
```

```
    MOV     pc, lr
```

```
full_buffer
```

```
    RSB     bitsfree, bitsfree, #0
```

```
    ORR     bitbuffer, bitbuffer, code, LSR bitsfree
```

```
    IF {ENDIAN} = "little"
```

```
        ;byte reverse the bit buffer prior to storing
```

```
        EOR     tmp, bitbuffer, bitbuffer, ROR #16
```

```
        AND     tmp, mask, tmp, LSR #8
```

```
        EOR     bitbuffer, tmp, bitbuffer, ROR #8
```

```
    ENDIF
```

```
    STR     bitbuffer, [bitstream], #4
```

```
    RSB     bitsfree, bitsfree, #32
```

```
    MOV     bitbuffer, code, LSL bitsfree
```

```
    MOV     pc, lr
```

```
bitstream_write_flush
```

```
    RSBS    bitsfree, bitsfree, #32
```

```
flush_loop
```

```
    MOVGT   bitbuffer, bitbuffer, ROR #24
```

```

STRGTB    bitbuffer, [bitstream], #1
SUBGTB    bitsfree, bitsfree, #8
BGT       flush_loop
MOV       pc, lr

```

6.7.3 可变宽度编码的位流解包

解开一个可变宽度编码的位流要比打包困难得多,主要是因为通常不知道正在解的位流中的编码宽度。对 Huffman 编码的位流,必须根据一串位的顺序来得到编码的长度并计算出相应的编码。

这里将使用查表的方法来提高解包的速度。这种方法是取位流的后续 N 位,然后在 2 个表 `look_codebits[]` 和 `look_code[]` 中进行查找,每一个表的大小是 2^N 个项。如果这 N 位可以决定编码,那么从这 2 个表中就可以分别得到编码长度和编码值。如果这 N 位不能够决定编码,那么表格 `look_codebits` 将会返回一个退出值 `0xff`,表示这种情况是例外。

在一个 Huffman 编码序列中,使用频率高的编码长度较短,使用频率低的编码长度较长。因此,可以使用查表的方法快速解码使用频率高的编码。在下面例子中,设 $N=8$,表的大小为 256 项。

【例 6.25】 提供 3 个函数来完成对一个大端的位流进行解包。

同例 6.24,这几个函数不符合 ATPCS 标准,一般需要使用内联函数。函数 `bitstream_read_start` 初始化程序,开始对位于字节地址 `bitstream` 的位流进行解码。每次调用函数 `bitstream_read_code`,就在寄存器 `code` 中返回下一个编码。这个函数只对能从查找表中读到的短编码进行处理,长编码在标号 `long_code` 处被处理,但这个功能的实现依赖于正在解码的具体要求。

这段代码使用了一个寄存器 `bitbuffer`,包含了从最高符号位开始的 $N + \text{bitsleft}$ 个编码位(见图 6.7)。



图 6.7 bitbuffer 的格式

<code>bitstream</code>	RN 0	;current byte address in the input bitstream
<code>look_code</code>	RN 2	;lookup table to convert next N - bits to a code
<code>look_codebits</code>	RN 3	;lookup table to convert next N - bits to a code length
<code>code</code>	RN 4	;code read


```

codebits    RN 5      ;length of code read
bitbuffer   RN 6      ;32 - bit input buffer (big endian)
bitsleft    RN 7      ;number of valid bits in the buffer - N
tmp         RN 8      ;scratch
tmp2        RN 9      ;scratch
mask        RN 12     ;N - bit extraction mask (1 << N) - 1

N           EQU 8      ;use a lookup table on 8 bits (N must be <= 9)

```

bitstream_read_start

```

        MOV     bitsleft, #32
read_fill_loop
        LDRB    tmp, [bitstream], #1
        ORR     bitbuffer, tmp, bitbuffer, LSL #8
        SUBS    bitsleft, bitsleft, #8
        BGT     read_fill_loop
        MOV     bitsleft, #(32 - N)
        MOV     mask, #(1 << N) - 1
        MOV     pc, lr

```

bitstream_read_code

```

        LDRB    codebits, [look_codebits, bitbuffer, LSR #(32 - N)]
        AND     code, mask, bitbuffer, LSR #(32 - N)
        LDR     code, [look_code, code, LSL #2]
        SUBS    bitsleft, bitsleft, codebits
        BMI     empty_buffer_or_long_code
        MOV     bitbuffer, bitbuffer, LSL codebits
        MOV     pc, lr

```

empty_buffer_or_long_code

```

        TEQ     codebits, #0xFF
        BEQ     long_code
        ;empty buffer - fill up with 3 bytes
        ;as N <= 9, we can fill 3 bytes without overflow
        LDRB    tmp, [bitstream], #1
        LDRB    tmp2, [bitstream], #1
        MOV     bitbuffer, bitbuffer, LSL codebits

```

```

LDRB    codebits, [bitstream], #1
ORR     tmp, tmp2, tmp, LSL#8
RSB     bitsleft, bitsleft, #(8-N)
ORR     tmp, codebits, tmp, LSL#8
ORR     bitbuffer, bitbuffer, tmp, LSL bitsleft
RSB     bitsleft, bitsleft, #(32-N)
MOV     pc, lr

```

```

long_code

```

```

;handle the long code case depending on the application
;here we just return a code of -1
MOV     code, #-1
MOV     pc, lr

```

计数器 bitsleft 实际上计算了在缓冲器 bitbuffer 中剩余的小于 N 位的下一次查表所需要的位的数目。只要 bitsleft ≥ 0 , 就可以进行下一次查表。一旦 bitsleft < 0 , 就会有两种可能: 一种可能是找到了一个有效的编码, 但没有足够的位来查找下一个编码; 另一种可能是 codebits 包含了退出值 0xff, 以指出编码长于 N 位。这两种情况都可以通过调用 empty_buffer_or_long_code 来捕获。如果缓冲器为空, 就对它填充 24 位。如果发现了一个长编码, 就跳转到 long_code 处。

在 ARM9TDMI 上, 该示例的最好性能是, 每个编码的解码只需要 7 个周期。如果能够提前知道编码打包时的位域宽度, 那么还可以获得更优的性能。

小 结 位操作

- ARM 可以使用逻辑操作和桶形移位器来高效地对位流进行打包和解包操作。
- 可以使用一个 32 位的寄存器作为位缓冲器来高效地访问位流, 再使用一个寄存器来保存位缓冲器中有效位的数目。
- 为了高效地对位流进行解码, 使用查表方法来扫描位流中的后续 N 位。通过查表可以对长度小于或等于 N 位的位流直接返回编码, 或者对长编码返回一个退出值。

6.8 高效的 switch

一条 switch 或多路分支语句表示在多个不同的动作之间进行选择。在这里假定这些不同的动作依赖于一个变量 x 。对不同的 x 值, 需要执行不同的动作。本节将讨论对不同类型的 x , 如何用汇编语言高效地实现 switch 结构。

6.8.1 在范围 $0 \leq x < N$ 的 switch

这个例子中 C 函数 `ref_switch` 根据不同的 x 值执行了不同的操作。这里只关心 x 的值在范围 $0 \leq x < 8$ 。

```
int ref_switch ( int x)
{
    switch (x)
    {
        case 0 ; return method_0 ();
        case 1 ; return method_1 ();
        case 2 ; return method_2 ();
        case 3 ; return method_3 ();
        case 4 ; return method_4 ();
        case 5 ; return method_5 ();
        case 6 ; return method_6 ();
        case 7 ; return method_7 ();
        default ; return method_d ();
    }
}
```

用 ARM 汇编来实现这种结构,有两种有效的方法。第一种方法是使用一个函数地址的表格,通过 x 的值从表格中索引并装载 `pc`。

【例 6.26】 `switch_absolute` 代码使用一个内联的函数指针表格实现 `switch` 结构。

```
x      RN 0

;int switch_absolute(int x)
switch_absolute
    CMP     x, #8
    LDRLT   pc, [pc, x, LSL#2]
    B       method_d
    DCD     method_0
    DCD     method_1
    DCD     method_2
    DCD     method_3
    DCD     method_4
```

```

DCD    method_5
DCD    method_6
DCD    method_7

```

由于 pc 寄存器是流水化操作的,所以这个程序可以正确执行。当 ARM 执行 LDR 指令时,pc 指向字 method_0。

上面的这种方法执行速度很快,但是有一个缺点:由于存储的是函数的绝对地址,这样代码就不是位置无关的。位置无关的代码通常是模块化的,并在运行时才被装入系统。下面的例子将介绍如何解决这个问题。

【例 6.27】 这里的代码 switch_relative 与上面的 switch_absolute 相比,执行速度相对慢一点。但是这段代码是位置无关的。

```

;int switch_relative(int x)
switch_relative
    CMP        x, #8
    ADDLT      pc, pc, x, LSL#2
    B          method_d
    B          method_0
    B          method_1
    B          method_2
    B          method_3
    B          method_4
    B          method_5
    B          method_6
    B          method_7

```

还有最后一个优化可以做。如果功能函数比较短,那么就可以直接使用内嵌指令来代替分支指令。

【例 6.28】 假定每一个跳转功能都由 4 条指令来实现。那么就可以用下面的代码:

```

CMP        x, #8
ADDLT      pc, pc, x, LSL#4    ;each method is 16 bytes long
B          method_d
Method_0
    ;the four instructions for method_0 go here
method_1
    ;the four instructions for method_1 go here
    ;... continue in this way...

```

6.8.2 基于通用变量 x 的 switch

现在假定 x 不是像 6.8.1 小节中那样在 0 到一个较小的范围内,那么如何才能在不轮流测试每种可能的 x 值的情况下,高效地实现 switch 结构呢?

在这种情况下,可以使用一种非常有用的技术,即散列(hashing)函数。散列函数可将任何函数 $y=f(x)$ 中我们感兴趣的值映射到一个连续的范围 $0 \leq y < N$ 。这样,可以把基于 x 的 switch,替换为基于 $y=f(x)$ 。如果出现冲突,就是如果 2 个 x 映射到同一个 y ,那么也会出现问题。这种情况下,就须使用额外的代码来测试可能导致冲突的所有可能的 x 的值。但是对我们这里的目的,一个好的散列函数应该是很容易计算得到的,不会出现很多冲突。

为了执行这种 switch,我们引入散列函数,并对散列值 y 使用 6.8.1 小节中优化过的 switch 代码。在 2 个 x 映射到同一个散列值 y 的地方,需要执行一个显式的测试;但对于一个好的散列函数,这种测试应该是罕见的。

【例 6.29】 假定对 8 种可能的 k ,当 $x=2^k$ 时,要调用 `method_k`。换句话说,我们希望 switch 发生在 x 值等于 1,2,4,8,16,32,64,128 时。对于其它的 x 值,只须调用一个默认的 `method_d` 函数。需要找一个 2 的幂减 1 的乘积形式的散列函数。尝试了不同的乘数,发现 $15 \times 31 \times x$ 对每一个 switch 值在位 9~11 有不同的值。这意味着可以使用这个乘积的第 9~11 位来作为我们的散列函数。

下面的汇编代码 `switch_hash` 使用这个散列函数执行这个 switch。注意其它不是 2 的幂的值将对应相同的散列值。这样 switch 语句的各种情况就转化成了一个简单的可以明确测试的 2 的幂的情况。如果 x 不是 2 的幂,那么就调用默认函数 `method_d`。

```

x      RN 0
hash   RN 1

;int switch_hash(int x)
switch_hash
    RSB    hash, x, x, LSL#4      ;hash = x * 15
    RSB    hash, hash, hash, LSL#5 ;hash = x * 15 * 31
    AND    hash, hash, #7 << 9   ;mask out the hash value
    ADD    pc, pc, hash, LSR#6
    NOP
    TEQ    x, #0x01
    BEQ    method_0
```

```

TEQ    x, #0x02
BEQ    method_1
TEQ    x, #0x40
BEQ    method_6
TEQ    x, #0x04
BEQ    method_2
TEQ    x, #0x80
BEQ    method_7
TEQ    x, #0x20
BEQ    method_5
TEQ    x, #0x10
BEQ    method_4
TEQ    x, #0x08
BEQ    method_3
B      method_d

```

小结 高效的 switch

- 对于 N 值比较小的情况,应确保 switch 的判断值 x 在范围 $0 \leq x < N$;也可以使用一个散列(hashing)函数来达到这个目的。
- 使用 switch 判断值来索引包含跳转函数指针的表格,或者跳转到有规则间隔的短代码段。第 2 种方法是位置无关的,而第 1 种方法不是(见 6.8.1 小节)。

6.9 边界不对齐数据的处理

如果 load 或 store 指令使用的地址不是传输数据宽度的倍数,那么就称为边界不对齐的数据访问。为了使代码对于不同的 ARM 体系结构和实现有良好的可移植性,应避免边界不对齐的数据访问。5.9 节介绍了用 C 实现对边界不对齐数据访问的解决方法。本节将讨论如何用汇编代码来处理边界不对齐的数据访问。

最简单的方法是使用一次只传送一字节数据的字节装载和存储。对于任何非速度敏感的操作,这是一种值得推荐的方法。下面的例子显示了如何使用这种方法来访问字变量。

【例 6.30】 显示使用边界不对齐的地址 p 来读/写一个 32 位字。

使用了 3 个临时寄存器 t0, t1 和 t2 来避免流水线互锁。所有边界不对齐的字操作,在 ARM9TDMI 上要占用 7 个周期。

注意: 对存储格式分别为大端和小端的 32 位数的操作要使用不同的函数处理。

```

p      RN 0
x      RN 1
t0     RN 2
t1     RN 3
t2     RN 12

```

```

;int load_32_little(char *p)

```

```

load_32_little

```

```

    LDRB    x,[p]
    LDRB    t0,[p, #1]
    LDRB    t1,[p, #2]
    LDRB    t2,[p, #3]
    ORR     x,x,t0,LSL#8
    ORR     x,x,t1,LSL#16
    ORR     r0,x,t2,LSL#24
    MOV     pc,lr

```

```

;int load_32_big(char *p)

```

```

load_32_big

```

```

    LDRB    x,[p]
    LDRB    t0,[p, #1]
    LDRB    t1,[p, #2]
    LDRB    t2,[p, #3]
    ORR     x,t0,x,LSL#8
    ORR     x,t1,x,LSL#8
    ORR     r0,t2,x,LSL#8
    MOV     pc,lr

```

```

;void store_32_little(char *p, int x)

```

```

store_32_little

```

```

    STRB    x,[p]
    MOV     t0,x,LSR#8
    STRB    t0,[p, #1]
    MOV     t0,x,LSR#16
    STRB    t0,[p, #2]
    MOV     t0,x,LSR#24
    STRB    t0,[p, #3]

```

```

MOV    pc, lr

;void store_32_big(char *p, int x)
store_32_big
MOV     t0, x, LSR#24
STRB    t0, [p]
MOV     t0, x, LSR#16
STRB    t0, [p, #1]
MOV     t0, x, LSR#8
STRB    t0, [p, #2]
STRB    x, [p, #3]
MOV     pc, lr

```

如果对每次访问要达到比 7 个周期更好的性能,那么就要编写以上程序的几个不同的变体,每个变体处理不同的地址边界情况。这样对边界不对齐数据的访问操作可以减少 3 个周期:1 条字装载和 2 条算术指令。

【例 6.31】 显示对于起始地址边界可能不对齐的字数据,生成 N 个字的校验和。这段代码是针对小端存储系统来编写的。注意如何使用汇编器保留字 MACRO 来生成 4 段程序 checksum_0,checksum_1,checksum_2 和 checksum_3。程序 checksum_a 处理了数据起始地址是 $4q+a$ 形式的情况。

使用宏定义可简化编程。我们只须编写一个宏,分 4 次来实现 4 个校验和程序段。

```

sum      RN 0      ;current checksum
N        RN 1      ;number of words left to sum
data     RN 2      ;word aligned input data pointer
w        RN 3      ;data word

;int checksum_32_little(char *data, unsigned int N)
checksum_32_little
BIC      data, r0, #3      ;aligned data pointer
AND      w, r0, #3        ;byte alignment offset
MOV      sum, #0          ;initial checksum
LDR      pc, [pc, w, LSL#2] ;switch on alignment
NOP                      ;padding
DCD      checksum_0
DCD      checksum_1
DCD      checksum_2

```



```

        DCD      checksum_3

        MACRO
        CHECKSUM $alignment
checksum_$alignment
        LDR      w, [data], #4      ;pre-load first value
10      ;loop
        IF $alignment<>0
            ADD    sum, sum, w, LSR#8 * $alignment
            LDR    w, [data], #4
            SUBS   N, N, #1
            ADD    sum, sum, w, LSL#32-8 * $alignment
        ELSE
            ADD    sum, sum, w
            LDR    w, [data], #4
            SUBS   N, N, #1
        ENDIF
        BGT      %BT10
        MOV      pc, lr
        MEND

        ;generate four checksum routines
        ;one for each possible byte alignment
        CHECKSUM 0
        CHECKSUM 1
        CHECKSUM 2
        CHECKSUM 3

```

现在可以像 6.6.2 小节那样展开和优化程序,以达到最高的执行速度。由于增加了代码量,一般只对实时性要求高的程序使用前述的优化技术。

小 结 边界不对齐数据的处理

- 如果程序的性能不成问题,那么可使用多字节装载和存储来访问边界不对齐的数据。这种方法可以访问给定字节排列方式的数据,而不管指针对齐和存储器系统的字节排列方式。
- 如果对程序的性能要求比较高,那么可以使用几段程序,对每一种不同的边界情况使用不同的程序段。可以使用汇编器保留字 MACRO 来自动生成这些程序段。

6.10 总 结

在一个应用程序中,要实现最好的性能,就需要编写优化的汇编程序。不过,只有对性能影响最大的关键程序才值得进行优化。可以使用性能分析器或者指令周期计数工具,比如 ARM 的 ARMulator Simulator,来找到这些敏感的关键程序段。

本章讨论的例子和使用的技术都是针对 ARM 汇编的。下面是一些关键的思想:

- 对代码进行合理的调整,这样就可以避免处理器流水线互锁或中止。附录 D 介绍了每一条指令产生结果所占用的时间。特别要注意 load 和乘法指令,这些指令产生结果往往需要更长的时间。
- 把尽可能多的数据存放在 14 个通用寄存器里,有时甚至应该把几个数据打包放入一个寄存器中。应避免把内层循环的变量放到堆栈中。
- 对于较小的 if 语句,使用条件数据处理操作比条件分支更好。
- 使用减计数到零(count - down - to - zero)的循环结构和循环展开的方法,以获得最高的循环执行效率。
- 对于打包和解包位流数据,使用 32 位的寄存器缓冲器可以提高效率,并可减小存储器数据带宽。
- 使用分支表格和散列(hash)函数来高效地实现 switch 语句。
- 为了高效地处理边界不对齐数据,可以使用多个程序段。针对输入和输出数组的特定边界情况,有多个不同程序段,并分别进行优化。运行时在其中选择一个程序段来执行。

第 7 章

基本运算优化

- 双精度整数乘法
- 整数规格化和前导 0 计数
- 除 法
- 平方根
- 超越函数: \log , \exp , \sin , \cos
- 字节顺序反转和位操作
- 饱和及舍入运算
- 随机数产生
- 总 结

基本运算指的是可以在各种不同的算法与程序中使用的操作。例如,加法、乘法、除法以及随机数发生等都是基本运算。一些基本运算是 ARM 指令集直接支持的,比如 32 位的加法和乘法。但是,许多基本运算并不被指令集直接支持,必须通过编写程序来实现这些基本运算(例如,除法和随机数产生)。

本章给出了一些常用基本运算的优化参考实现。前 3 节是关于乘法和除法运算的;7.1 节分析了扩展精度乘法的实现方法;7.2 节讨论了规格化操作,这对于 7.3 节中的除法运算是非常有用的。

接下来的 2 节介绍了一些更加复杂的数学运算;7.4 节讨论的是平方根的问题;7.5 节则介绍了超越函数,如 \log , \exp , \sin 和 \cos 的实现方法;7.6 节讨论的是位运算;7.7 节介绍饱和运算与舍入(四舍五入)操作;最后,7.8 节描述了随机数产生问题。

可以用两种方式来阅读、使用本章的内容。首先,可以把它作为一种直接的参考。如果需要一段除法程序,可以直接通过目录找到这个程序,或者找到除法这一小节。读者可以从本书的网站上拷贝到汇编源代码。其次,本章提供了一些理论知识,用以解释各种实现方法的原理,这些理论对于修改和扩展这些程序是必需的。例如,对于一种运算的输入/输出数据的精度、格式可能有不同的要求。正因为如此,在一些文字描述中,可能包含许多数学公式和一些枯燥的证明过程。如果对这些不感兴趣,可以忽略跳过。

可以从本书的网站上直接下载到完整的示例代码,并且可以使用 ARM 提供的工具直接进行编译或汇编。为了保持一致,本章中的所有例子使用的工具是 ARM 工具包 ADS1.1。在附录 A.4 中可以找到关于汇编器格式的帮助信息。同样,也可以使用 GUN 的汇编器 `gas`,在附录 A.5 中可以找到关于它的格式的帮助信息。

本章使用了 C 语言的关键字 `_value_in_regs`。在 ARM 的编译器 `armcc` 中,它表示函数的参数或返回值应该通过寄存器而不是引用来进行传递。在实际应用中,这将不是一个问题,因为操作将被内联,以提高运算效率。

本章中,使用记号 Q_k 表示一个定点数,其二进制小数点位于 $k-1$ 位和 k 位之间。例如 0.75 以 Q_{15} 表示,就是整数值 $0x6000$ 。关于 Q_k 表示形式和定点算法的更多细节,请参见 8.1 节。“以 Q_{15} 形式表示的 $d < 0.5$ ”的意思是, d 表示的值是 $d2^{-15}$,这个值小于 $1/2$ 。

7.1 双精度整数乘法

使用 `UMULL` 和 `SMULL` 指令可以进行最多 32 位宽度的整数乘法运算。下面的程序实现了 64 位有/无符号整数相乘,得到一个 64 位或者 128 位的结果。按同样的方法,可以进行扩展,从而实现任何长度的整数乘法运算。有许多需要长整型乘法运算的应用,例如,处理 C 中 `long long` 类型数据,仿真双精度定点或浮点运算以及公共密钥计算需要的长整

型算术运算等。

对于多字数据,这里使用小端字节排列方式表示。比如一个 128 位的整数存放在 4 个寄存器 a_3 , a_2 , a_1 和 a_0 , 相应的存储位是 $[127:96]$, $[95:64]$, $[63:32]$, $[31:0]$ (见图 7.1)。

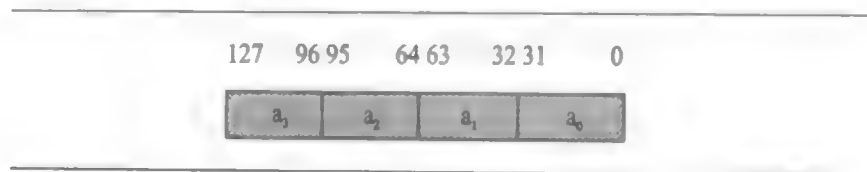


图 7.1 4 个 32 位值表示的一个 128 位数据

7.1.1 长整型乘法

使用下面的 3 条指令序列来完成 2 个 64 位数(有符号或无符号) b 和 c 的乘法, 并得到一个 64 位的 long long 类型的结果 a 。不包括 ARM Thumb 函数调用规则(ATPCS)的封装开销和最坏情况的输入值, 这个运算在 ARM7TDMI 上会占用 24 个周期; 在 ARM9TDMI 上占用 25 个周期; 在 ARM9E 上占用 8 个周期。在这些周期中, 有一个是由于第一条 UMULL 指令和 MLA 指令之间的流水线互锁造成的, 这可以通过与其它代码的交错使用来避免。

```

b_0    RN 0    ;b bits [31:00] (b low)
b_1    RN 1    ;b bits [63:32] (b high)
c_0    RN 2    ;c bits [31:00] (c low)
c_1    RN 3    ;c bits [63:32] (c high)
a_0    RN 4    ;a bits [31:00] (a low-low)
a_1    RN 5    ;a bits [63:32] (a low-high)
a_2    RN 12   ;a bits [95:64] (a high-low)
a_3    RN lr   ;a bits [127:96] (a high-high)

```

```

;long long mul_64to64(long long b, long long c)
mul_64to64
    STMFD    sp!, {r4,r5,lr}
    ;64-bit a = 64-bit b * 64-bit c
    UMULL    a_0, a_1, b_0, c_0    ;low * low
    MLA      a_1, b_0, c_1, a_1    ;low * high
    MLA      a_1, b_1, c_0, a_1    ;high * low
    ;return wrapper
    MOV      r0, a_0

```

```
MOV    r1, a_1
LDMFD  sp!, {r4,r5,pc}
```

7.1.2 128 位结果的无符号 64 位乘法

对于 2 个无符号 64 位数相乘得到 128 位结果的运算,有 2 种略微不同的实现方法。在 ARM7M 版本上,第一种方法执行比较快。与无累加的版本相比,这里的乘累加指令占用了一个额外的周期。在 ARM7M 版本上,需要 4 个长乘法和 6 个加法,最差情况下需要 30 个周期。

```
    __value_in_regs struct { unsigned a0,a1,a2,a3;}
    ; umul_64to128_arm7m(unsigned long long b,
    ;                      unsigned long long c)
umul_64to128_arm7m
    STMFD  sp!, {r4,r5,lr}
    ;unsigned 128-bit a = 64-bit b * 64-bit c
    UMULL  a_0, a_1, b_0, c_0          ;low * low
    UMULL  a_2, a_3, b_0, c_1          ;low * high
    UMULL  c_1, b_0, b_1, c_1          ;high * high
    ADDS   a_1, a_1, a_2
    ADCS  a_2, a_3, c_1
    ADC    a_3, b_0, #0
    UMULL  c_0, b_0, b_1, c_0          ;high * low
    ADDS   a_1, a_1, c_0
    ADCS  a_2, a_2, b_0
    ADC    a_3, a_3, #0
    ;return wrapper
    MOV    r0, a_0
    MOV    r1, a_1
    MOV    r2, a_2
    MOV    r3, a_3
    LDMFD  sp!, {r4,r5,pc}
```

在 ARM9TDMI 和 ARM9E 版本上,下面的第二种方法执行效果比较好。这里乘累加的执行速度和普通乘法一样快。在 ARM9E 版本上,调整好乘法指令,可避免数据结果相关而导致的流水线互锁现象(有关流水线和互锁的详细介绍见 6.2 节)。

```
    __value_in_regs struct { unsigned a0,a1,a2,a3;}
```

```

; umul_64tol28_arm9e(unsigned long long b,
;                      unsigned long long c)
umul_64tol28_arm9e
    STMFD    sp!, {r4,r5,lr}
; unsigned 128-bit a = 64-bit b * 64-bit c
    UMULL    a_0, a_1, b_0, c_0      ; low * low
    MOV      a_2, #0
    UMLAL    a_1, a_2, b_0, c_1      ; low * high
    MOV      a_3, #0
    UMLAL    a_1, a_3, b_1, c_0      ; high * low
    MOV      b_0, #0
    ADDS     a_2, a_2, a_3
    ADC      a_3, b_0, #0
    UMLAL    a_2, a_3, b_1, c_1      ; high * high
; return wrapper
    MOV      r0, a_0
    MOV      r1, a_1
    MOV      r2, a_2
    MOV      r3, a_3
    LDMFD    sp!, {r4,r5,pc}

```

不包括函数调用和返回的封装开销,这个实现在 ARM9TDMI 上需要 33 个周期,在 ARM9E 上需要 17 个周期。这种思想是:如果 a, b, c 和 d 都是无符号 32 位整数,那么操作 $ab+c+d$ 不会超过一个 64 位无符号整数的范围。因此可以使用通常教科书上操作 $ab+c+d$ 的方法来实现长整型乘法,这里 c 和 d 是横向和纵向的进位。

7.1.3 128 位结果的有符号 64 位整数乘法

一个有符号的 64 位整数可以分解为一个有符号的高 32 位和一个无符号的低 32 位。为了实现 b 的高位部分和 c 的低位部分相乘,需要一条有符号数和无符号数相乘的指令。尽管 ARM 没有这样的指令,但可以用宏来合成一条这样的指令。

下面的宏 USMLAL 提供了一个无符号数和有符号数乘累加的操作。要实现无符号数 b 和有符号数 c 相乘,首先需把 2 个数都当成有符号数相乘产生的结果 bc 。如果 b 的最高位为 1,那么这个有符号乘法再与 $(b-2^{32})$ 相乘。在这种情况下,通过对结果与 $c2^{32}$ 相加来校正结果值。类似地, SUMLAL 实现了一个有符号数与无符号数的乘累加运算。

```

MACRO
    USMLAL $al, $ah, $b, $c
    ;signed $ah. $al += unsigned $b * signed $c
    SMLAL   $al, $ah, $b, $c      ;a = (signed)b * c;
    TST     $b, #1 << 31          ;if ((signed)b<0)
    ADDNE   $ah, $ah, $c          ;a += (c << 32);
MEND

MACRO
    SUMLAL $al, $ah, $b, $c
    ;signed $ah. $al += signed $b * unsigned $c
    SMLAL   $al, $ah, $b, $c      ;a = b * (signed)c;
    TST     $c, #1 << 31          ;if ((signed)c<0)
    ADDNE   $ah, $ah, $b          ;a += (b << 32);
MEND

```

使用这些宏,把 7.1.2 小节的 64 位乘法转换成有符号的乘法要相对简单一些。在相同应用环境中,有符号的乘法要比其相应的无符号乘法多 4 个周期,这是因为有符号数和无符号数相乘需要额外的用于符号纠正的指令。

```

    __value_in_regs struct { unsigned a0,a1,a2;signed a3;}
    ; smul_64to128(long long b, long long c)
smul_64to128
    STMFD   sp!, {r4,r5,lr}
    ;signed 128-bit a = 64-bit b * 64-bit c
    UMULL   a_0, a_1, b_0, c_0      ;low * low
    MOV     a_2, #0
    USMLAL  a_1, a_2, b_0, c_1      ;low * high
    MOV     a_3, #0
    SUMLAL  a_1, a_3, b_1, c_0      ;high * low
    MOV     b_0, a_2, ASR#31
    ADDS    a_2, a_2, a_3
    ADC     a_3, b_0, a_3, ASR#31
    SMLAL   a_2, a_3, b_1, c_1      ;high * high
    ;return wrapper
    MOV     r0, a_0
    MOV     r1, a_1
    MOV     r2, a_2
    MOV     r3, a_3
    LDMFD   sp!, {r4,r5,pc}

```


7.2 整数规格化和前导 0 计数

当一个整数的前导 1 或最高有效位所在的位置为规定的标准值时,这个整数就被规格化了。实现 Newton-Raphson 除法(见 7.3.2 小节)或把数转换成浮点格式,都将用到整数的规格化。规格化对于对数运算(见 7.5.1 小节)和一些调度(dispatch)程序用到的优先解码器也是很有帮助的。在这些应用中,须知道规格化后的值和得到此值的移位值。

这个操作是如此重要,以至在 ARMv5E 以上的体系结构中,特别新增了一条指令来加速规格化操作。CLZ 指令用来计算第一个有效位 1 之前的前导 0(leading zero)的个数,如果一个 1 都没有,则返回 32。CLZ 的值是需要规格化的整数将要左移的值,这样规格化后的前导 1 就在位 31。

7.2.1 ARMv5 及以上体系结构的整数规格化

在 ARMv5 体系结构的平台上,下面的代码分别完成了无符号和有符号数的规格化。无符号数规格化通过左移使得最高有效位(前导 1)处于第 31 位;而有符号数则通过左移直到符号位处于第 31 位,前导数据位(leading bit)处于第 30 位。这 2 个函数都返回一个存放在寄存器中的结构体,结构体中的 2 个值分别为规格化后的整数和规格化时左移的位数。

```
x      RN 0      ;input, output integer
shift  RN 1      ;shift to normalize

;__value_in_regs struct { unsigned x,int shift;}
;  unorm_arm9e(unsigned x)
unorm_arm9e
    CLZ      shift, x          ;left shift to normalize
    MOV      x, x, LSL shift   ;normalize
    MOV      pc, lr

;__value_in_regs struct { signed x,int shift;}
;  unorm_arm9e(signed x)
snorm_arm9e                                ;[s s s 1 - s x x...]
    EOR      shift, x, x, LSL#1 ;[0 0 1  x  x x...]
    CLZ      shift, shift      ;left shift to normalize
```

ARM 嵌入式系统开发

```

MOV    x, x, LSL shift      ;normalize
MOV    pc, lr

```

注意：上面的程序中，使用了逻辑“异或”指令，把对有符号数规格化的规则简化为无符号数的规则。如果 x 是有符号数，那么通过 $x(x \ll 1)$ ，就把前导 1 放在了 x 的第一个符号位置上。

7.2.2 在 ARMv4 体系结构上的规格化

ARMv4 体系结构的处理器，比如 ARM7TDMI 或 ARM9TDMI，不支持 CLZ 指令，需要使用其它的方法来实现这个功能。在函数 `unorm_arm7m` 中使用的二分法(divide-and-conquer)较好地平衡了性能和代码量之间的关系，通过依次检查对 x 的 16,8,4,2,1 位左移是否可以进行，来完成规格化。

```

    _value_in_regs struct { unsigned x; int shift; }
    ; unorm_arm7m(unsigned x)
unorm_arm7m
    MOV    shift, #0          ;shift = 0;
    CMP    x, #1 << 16        ;if (x < (1 << 16))
    MOVOC   x, x, LSL #16      ;{ x = x << 16;
    ADDCC   shift, shift, #16   ; shift += 16; }
    TST    x, #0xFF000000      ;if (x < (1 << 24))
    MOVEQ   x, x, LSL #8       ;{ x = x << 8;
    ADDEQ   shift, shift, #8    ; shift += 8; }
    TST    x, #0xF0000000      ;if (x < (1 << 28))
    MOVEQ   x, x, LSL #4       ;{ x = x << 4;
    ADDEQ   shift, shift, #4    ; shift += 4; }
    TST    x, #0xC0000000      ;if (x < (1 << 30))
    MOVEQ   x, x, LSL #2       ;{ x = x << 2;
    ADDEQ   shift, shift, #2    ; shift += 2; }
    TST    x, #0x80000000      ;if (x < (1 << 31))
    ADDEQ   shift, shift, #1    ;{ shift += 1;
    MOVEQS  x, x, LSL #1       ; x << = 1;
    MOVEQ   shift, #32         ;if (x == 0) shift = 32; }
    MOV    pc, lr

```

上面的程序中，当 x 为 0 时，最后的 MOVEQ 指令设置 shift 为 32，这一步经常可以省略。在 ARM7TDMI 或 ARM9TDMI 上，整个实现需要 17 个周期，足以满足大部分的应用需求。但是，对这些处理器而言，这还不是最快速的规格化方法。想要获得最快的执行速

度,可以采用基于散列算法(hash-based)的方法。

基于散列算法的方法,首先是在不改变 CLZ 值(前导 0 个数)的情况下,把输入操作数缩小到 33 种不同的可能之一。实现方式为:首先需要重复 $x = x \mid (x \gg s)$ 操作, $s = 1, 2, 4, 8$, 当前导 1 的初始位置在位 31~15 之间时,这样就把前导 1 向右复制了 16 位。当前导 1 的初始位置在位 0~14 之间时,前导 1 就从初始位置复制到位 0。移位操作结束之后,须计算 $x = x \& \sim(x \gg 16)$,这是为了清除最后一个复制 1 到位 0 之间的位。表 7.1 显示了这 2 步操作的效果,对每一种可能的二进制输入形式,显示了这些操作产生的对应 32 位编码。

注意:输入码的 CLZ 值和 32 位编码的 CLZ 值是相同的。

表 7.1 不同输入值的编码和 CLZ 值

Input(in binary, x is a wildcard bit)				32-bit code	CLZ value
1xxxxxxx	xxxxxxx	xxxxxxx	xxxxxxx	0xFFFF0000	0.
01xxxxxx	xxxxxxx	xxxxxxx	xxxxxxx	0x7FFF8000	1
001xxxxx	xxxxxxx	xxxxxxx	xxxxxxx	0x3FFFC000	2
...			
00000000	00000000	00000000	000001xx	0x00000007	29
00000000	00000000	00000000	0000001x	0x00000003	30
00000000	00000000	00000000	00000001	0x00000001	31
00000000	00000000	00000000	00000000	0x00000000	32

通过使用散列函数和查表相结合的方法,可以得到每个编码 CLZ 值。关于散列函数的细节可参见 6.8.2 小节。

对于散列函数,可通过与一个很大的值相乘,截取结果的最前面 6 位。在 ARM 上,使用桶形移位器可以很容易地进行 $2^s + 1$ 和 $2^s - 1$ 形式的乘法运算。事实上,对每个不同的 CLZ 值,乘以 $(2^9 - 1)(2^{11} - 1)(2^{14} - 1)$ 会得到不同的散列值。作者通过计算机计算搜索,找到了这个乘数。

在 ARMv4 处理器上,下面的代码实现了一个快速的基于散列算法的规格化。在 ARM7TDMI 上,整个执行过程除了建立表指针外,需要 13 个周期。

```

table    RN 2      ;address of hash lookup table

        ;_value_in_regs struct { unsigned x;int shift;}
        ; unorm_arm7m_hash(unsigned x)
unorm_arm7m_hash
        ORR      shift, x, x, LSR#1

```

```

    ORR    shift, shift, shift, LSR#2
    ORR    shift, shift, shift, LSR#4
    ORR    shift, shift, shift, LSR#8
    BIC    shift, shift, shift, LSR#16
    RSB    shift, shift, shift, LSL#9    ; * (29 - 1)
    RSB    shift, shift, shift, LSL#11   ; * (211 - 1)
    RSB    shift, shift, shift, LSL#14   ; * (214 - 1)
    ADR    table, unorm_arm7m_hash_table
    LDRB    shift, [table, shift, LSR#26]
    MOV    x, x, LSL shift
    MOV    pc, lr

```

unorm_arm7m_hash_table

```

    DCB    0x20, 0x14, 0x13, 0xff, 0xff, 0x12, 0xff, 0x07
    DCB    0x0a, 0x11, 0xff, 0xff, 0x0e, 0xff, 0x06, 0xff
    DCB    0xff, 0x09, 0xff, 0x10, 0xff, 0xff, 0x01, 0x1a
    DCB    0xff, 0x0d, 0xff, 0xff, 0x18, 0x05, 0xff, 0xff
    DCB    0xff, 0x15, 0xff, 0x08, 0x0b, 0xff, 0x0f, 0xff
    DCB    0xff, 0xff, 0xff, 0x02, 0x1b, 0x00, 0x19, 0xff
    DCB    0x16, 0xff, 0x0c, 0xff, 0xff, 0x03, 0x1c, 0xff
    DCB    0x17, 0xff, 0x04, 0x1d, 0xff, 0xff, 0x1e, 0x1f

```

7.2.3 后缀 0 计数

后缀 0 计数与前导 0 计数紧密相关。后缀 0 计数表示了一个整数中最低有效位后的 0 的个数，它也是可以被该整数除的最大的 2 的乘幂。因此可以通过计算后缀 0 的个数，把一个整数表示成 2 的乘幂与一个奇整数的积。如果一个整数为 0，那么它没有最低有效位，这个后缀 0 的个数就是 32。

为了找到非零整数 n 可除的最大 2 的乘幂，有一个巧妙的方法，即表达式 $(n \& (-n))$ 值中惟一的一个 1 的位置，就是 n 中最低有效位 1 的位置。图 7.2 显示了这个过程。 x 表示通配符位。

用这种方法，可以把后缀 0 计数转换成前导 0 计数。下面的代码实现了在 ARM9E 上计算后缀 0 个数的功能。这里使用了条件指令，在不增加其它额外开销的情况下，解决了输入操作数是 0 的问题。

```

n = xxxxxxxxxxxxxxxxxxxx10000000000000
-n = yyyyyyyyyyyyyyyyyyy10000000000000 其中 y=1-x
n & (-n) = 000000000000000001000000000000

```

图 7.2 识别最低有效位

```

;unsigned ctz_arm9e(unsigned x)
ctz_arm9e
    RSBS    shift, x, #0      ;shift = -x
    AND     shift, shift, x    ;isolate trailing 1 of x
    CLZCC   shift, shift      ;number of zeros above last 1
    RSC     r0, shift, #32     ;number of zeros below last 1
    MOV     pc, lr

```

对于不支持 CLZ 指令的处理器来说,类似于 7.2.2 小节的基于散列算法的方法可以得到较好的性能:

```

;unsigned ctz_arm7m(unsigned x)
ctz_arm7m
    RSB     shift, x, #0
    AND     shift, shift, x    ;isolate lowest bit
    ADD     shift, shift, shift, LSL#4 ; * (2^4 + 1)
    ADD     shift, shift, shift, LSL#6 ; * (2^6 + 1)
    RSB     shift, shift, shift, LSL#16 ; * (2^16 - 1)
    ADR     table, ctz_arm7m_hash_table
    LDRB    r0, [table, shift, LSR#26]
    MOV     pc, lr

ctz_arm7m_hash_table
    DCB     0x20, 0x00, 0x01, 0x0c, 0x02, 0x06, 0xff, 0x0d
    DCB     0x03, 0xff, 0x07, 0xff, 0xff, 0xff, 0xff, 0x0e
    DCB     0x0a, 0x04, 0xff, 0xff, 0x08, 0xff, 0xff, 0x19
    DCB     0xff, 0xff, 0xff, 0xff, 0xff, 0x15, 0x1b, 0x0f
    DCB     0x1f, 0x0b, 0x05, 0xff, 0xff, 0xff, 0xff, 0xff
    DCB     0x09, 0xff, 0xff, 0x18, 0xff, 0xff, 0x14, 0x1a
    DCB     0x1e, 0xff, 0xff, 0xff, 0xff, 0x17, 0xff, 0x13
    DCB     0x1d, 0xff, 0x16, 0x12, 0x1c, 0x11, 0x10

```

7.3 除 法

ARM 核对除法运算不提供硬件支持。除法运算必须通过调用基于标准算术操作的软件程序来完成。即使在程序中不能避免除法(如何避免除法运算以及相同除数的快速除法运算见 5.10 节),也应该使用经过高度优化的除法程序。本节将提供几个有用的经过优化的除法程序。

对于最大限度优化的 newton - raphson 除法程序来说,在 ARM9E 上的运行速度,可以和每周期 1 位的硬件除法的执行速度一样快。所以,ARM 并不需要复杂的硬件除法实现。

本节描述了一些目前最快速的除法实现。其篇幅不可避免地有点长,因为有许多不同的除法技术和精度需要考虑。另外,需要证明程序对所有可能的输入值都能正确执行。考虑到不可能——尝试所有可能的 32 位数除法的输入参数,所以这些证明是必要的。如果对理论细节不感兴趣,可以跳过有关证明部分,直接从书中获取代码。

7.3.1 小节介绍了使用试探减法(trial subtraction)或二分搜索(binary search)的除法实现。对于可能由小的商引起的早先结束(*early termination*)的除法,或者是处理器不支持快速乘法指令,试探减法是很有用的。7.3.2 小节和 7.3.3 小节给出了使用 Newton - Raphson 迭代方法实现的除法程序。当最坏情况下的性能是重要的,或者有快速的乘法指令支持时,可以使用 Newton - Raphson 迭代。Newton - Raphson 实现使用了 ARMV5TE 扩展。7.3.4 小节着重于分析有符号数的除法。

本章使用下面的符号,以区分精确的数学意义上的除法和整型除法运算:

- n/d 表示 n 除以 d 的整数商,结果趋向于 0(与 C 中相同);
- $n\%d$ 表示 n 除以 d 的整数余数,就是 $n - d(n/d)$ (与 C 中相同);
- $n//d = \frac{n}{d} = nd^{-1}$ 表示真正数学意义上的 n 被 d 除。

7.3.1 通过试探减法实现无符号数除法

计算无符号整数 n 和 d 的商 $q = n/d$ 和余数 $r = n\%d$ 。假定已知商 q 不超过 N 位, $n/d < 2^N$,或者说 $n < (d \ll N)$ 。试探减法算法以最高有效位——位 $N-1$ 开始,通过依次试图设置各个位来确定 q 的 N 位值。这种做法和对结果的二分搜索(binary search)是等价的。如果可以从当前的余数减去($d \ll k$),且减法结果不是负数,那么就可以设置位 k 。例子 `udiv_simple` 给出了这个算法的 C 实现代码:

```

unsigned udiv_simple(unsigned d, unsigned n, unsigned N)
{
    unsigned q = 0, r = n;

    do
    {
        /* calculate next quotient bit */
        N--; /* move to next bit */
        if ((r >> N) >= d) /* if r >= d * (1 << N) */
        {
            r -= (d << N); /* update remainder */
            q += (1 << N); /* update quotient */
        }
    } while (N);

    return q;
}

```

【证明 7.1】 对上面结果的正确性做出证明。

注意：在减 N 之前，等式(7.1)的不变式成立。

$$n = qd + r \quad \text{且} \quad 0 \leq r < d2^N \quad (7.1)$$

开始时, $q=0, r=n$, 由于假设商的范围在 N 位内, 因而不变式成立。假定现在对某个 N , 不变式成立。如果 $r < d2^{N-1}$, 那么对于 $N-1$, 不需要做任何事情, 不变式成立。如果 $r \geq d2^{N-1}$, 那么通过对 r 减去 $d2^{N-1}$, 并加 2^{N-1} 到 q , 可保持不变式成立。

前面的实现被称为恢复(restoring)试探减法实现。在非恢复实现中, 减法总会发生。但是, 如果 r 变成了负数, 则要在下一次进行 $(d \ll N)$ 的加法, 而不是减法, 这样能得到相同的结果。非恢复除法在 ARM 上执行速度比较慢, 因此就不再详细讨论了。下面将给出使用试探减法的除法汇编实现, 它可以针对不同大小的除数和被除数。这些程序可以在所有的 ARM 处理器上运行。

7.3.1.1 通过试探减法实现的无符号 32 位/32 位除法

这是一个 C 编译器所需要的操作。当在 C 中出现表达式 n/d 或 $n\%d$, 而且 d 又不是 2 的乘幂时, C 编译器就需要调用它。程序返回一个带有商和余数 2 个元素的结构体。

d	RN 0	;input denominator d, output quotient
r	RN 1	;input numerator n, output remainder
t	RN 2	;scratch register
q	RN 3	;current quotient

```

    ;_value_in_regs struct { unsigned q, r;}
    ;   udiv_32by32_arm7m(unsigned d, unsigned n)

udiv_32by32_arm7m
    MOV     q, #0                ;zero quotient
    RSBS    t, d, r, LSR#3       ;if ((r >> 3) >= d) C=1;else C=0;
    BCC     div_3bits            ;quotient fits in 3 bits
    RSBS    t, d, r, LSR#8       ;if ((r >> 8) >= d) C=1;else C=0;
    BCC     div_8bits            ;quotient fits in 8 bits
    MOV     d, d, LSL#8          ;d = d * 256
    ORR     q, q, #0xFF000000    ;make div_loop iterate twice
    RSBS    t, d, r, LSR#4       ;if ((r >> 4) >= d) C=1;else C=0;
    BCC     div_4bits            ;quotient fits in 12 bits
    RSBS    t, d, r, LSR#8       ;if ((r >> 8) >= d) C=1;else C=0;
    BCC     div_8bits            ;quotient fits in 16 bits
    MOV     d, d, LSL#8          ;d = d * 256
    ORR     q, q, #0x00FF0000    ;make div_loop iterate 3 times
    RSBS    t, d, r, LSR#8       ;if ((r >> 8) >= d)
    MOVCS   d, d, LSL#8          ;{ d = d * 256;
    ORRCS   q, q, #0x0000FF00    ;make div_loop iterate 4 times}
    RSBS    t, d, r, LSR#4       ;if ((r >> 4) < d)
    BCC     div_4bits            ;r/d quotient fits in 4 bits
    RSBS    t, d, #0             ;if (0 >= d)
    BCS     div_by_0             ;goto divide by zero trap
    ;fall through to the loop with C=0

div_loop
    MOVCS   d, d, LSR#8          ;if (next loop) d = d/256
div_8bits
    ;calculate 8 quotient bits
    RSBS    t, d, r, LSR#7       ;if ((r >> 7) >= d) C=1;else C=0;
    SUBCS   r, r, d, LSL#7       ;if (C) r -= d << 7;
    ADC     q, q, q              ;q = (q << 1) + C;
    RSBS    t, d, r, LSR#6       ;if ((r >> 6) >= d) C=1;else C=0;
    SUBCS   r, r, d, LSL#6       ;if (C) r -= d << 6;
    ADC     q, q, q              ;q = (q << 1) + C;
    RSBS    t, d, r, LSR#5       ;if ((r >> 5) >= d) C=1;else C=0;
    SUBCS   r, r, d, LSL#5       ;if (C) r -= d << 5;
    ADC     q, q, q              ;q = (q << 1) + C;
    RSBS    t, d, r, LSR#4       ;if ((r >> 4) >= d) C=1;else C=0;

```



```

        SUBCS    r, r, d, LSL#4      ;if (C) r -= d << 4;
        ADC      q, q, q              ;q = (q << 1) + C;
div_4bits                                ;calculate 4 quotient bits
        RSBS     t, d, r, LSR#3      ;if ((r >> 3) >= d) C=1;else C=0;
        SUBCS    r, r, d, LSL#3      ;if (C) r -= d << 3;
        ADC      q, q, q              ;q = (q << 1) + C;
div_3bits                                ;calculate 3 quotient bits
        RSBS     t, d, r, LSR#2      ;if ((r >> 2) >= d) C=1;else C=0;
        SUBCS    r, r, d, LSL#2      ;if (C) r -= d << 2;
        ADC      q, q, q              ;q = (q << 1) + C;
        RSBS     t, d, r, LSR#1      ;if ((r >> 1) >= d) C=1;else C=0;
        SUBCS    r, r, d, LSL#1      ;if (C) r -= d << 1;
        ADC      q, q, q              ;q = (q << 1) + C;
        RSBS     t, d, r              ;if (r >= d) C=1;else C=0;
        SUBCS    r, r, d              ;if (C) r -= d;
        ADCS     q, q, q              ;q = (q << 1) + C; C=old q bit 31;
div_next
        BCS      div_loop            ;loop if more quotient bits
        MOV      r0, q                ;r0 = quotient; r1 = remainder;
        MOV      pc, lr               ;return { r0, r1 } structure;
div_by_0
        MOV      r0, #-1
        MOV      r1, #-1
        MOV      pc, lr               ;return { -1, -1 } structure;

```

为了弄清楚这个程序段是如何工作的,首先观察标号 `div_8bits` 和 `div_next` 之间的代码。其计算了 8 位的商 n/d , 把余数放在 r , 并把 8 位的商插入到 q 的低位上。这段代码使用了试探减法来完成, 依次试图对 r 减去 $128d, 64d, 32d, 16d, 8d, 4d, 2d$ 和 d 。对每一次减法, 如果减法的结果不小于零, 则设进位为 1; 否则设为 0。这个进位构成了要插入 q 的结果的下一位。

接下来, 如果只要执行 4 位或者 3 位的除法运算, 则可以分别跳到 `div_4bits` 或 `div_3bits` 的代码段。

现在来看一下程序的开始。计算 r/d , 把余数放在 r , 并把商写到 q 。首先检查商 q 是否在 3 或 8 位之内。如果是, 就可以直接跳转到 `div_3bits` 或 `div_8bits` 程序段, 分别来计算相应的结果。当商是比较小的值时, 这种早先结束 (*early termination*) 是很有用的。如果商超过 8 位, 那么就对 d 乘以 256, 直到 r/d 的结果在 8 位之内。同时, 把对 d 乘以 256 的次数记录下来, 放在 q 的高位, 对每一次乘法设置 8 位。这就意味着在计算 8 位 r/d 之后, 又回

7.3.1.2 通过试探减法实现的无符号 32 位/15 位除法

下面的代码,被除数 N 是一个 32 位的无符号整数,除数 d 是一个 15 位的无符号整数。程序返回一个包含 15 位商 q 和余数 r 的结构体。如果 $n \geq (d < 15)$,那么结果就会溢出,这时将返回可允许的最大商值 $0x7fff$ 。

212

```

ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
ADCS    r, m, r, LSL #1
SUBCC   r, r, m
;extract answer and remainder (if required)
ADC     r0, r, r           ;insert final answer bit
MOV     r, r0, LSR #15     ;extract remainder
BIC     r0, r0, r, LSL #15 ;extract quotient
MOV     pc, lr             ;return { r0, r }
overflow_15                  ;quotient overflows 15 bits
LDR     r0, = 0x7FFF        ;maximum quotient
MOV     r1, r0              ;maximum remainder
MOV     pc, lr              ;return { 0x7fff, 0x7fff }

```

首先设置 $m = -d2^{14}$ 。用移位后的余数加上负的除数来代替从余数减去一个移位后的除数。在经过第 k 次的试探减法后, r 低端的 k 位就存放了商的最高 k 位; r 高端的 $(32-k)$ 位存放了余数值。每一条 ADC 指令实现 3 个功能:

- 对余数左移一位;
- 从上一次的试探减法来增加一位商值;
- 从余数减去 $(d \ll 14)$ 。

经过 15 次后, r 的低端 15 位就包含了商的值, 高端 17 位包含了余数值。把这 2 个值分别放进寄存器 $r0$ 和 $r1$ 。除了返回之外, 这个除法在 ARM7TDMI 上共占用了 35 个周期。

7.3.1.3 通过试探减法实现的无符号 64 位/31 位除法

这种操作适合于 Q31 形式的定点整数除法运算(见 8.1.5 小节)。这比 7.3.1.2 小节中的除法运算精度提高了 1 倍。被除数 n 是一个 64 位的无符号整数,除数 d 是一个 31 位的无符号整数。下面的程序返回一个包含 32 位商 q 和余数 r 的结构体。如果 $n \geq d2^{32}$,那么结果会溢出。此时程序将返回可允许的最大商值 0xffffffff。这个使用每周期 3 位(three-bit-per-cycle)的试探减法的除法程序,在 ARM7TDMI 上共占用 99 个周期。在代码的注释中,使用符号 $[r, q]$ 表示一个高 32 位 r 和低 32 位 q 的 64 位值。

```

m      RN 0      ;input denominator d, -d
r      RN 1      ;input numerator (low), remainder (high)
t      RN 2      ;input numerator (high)
q      RN 3      ;result quotient and remainder (low)

;_value_in_regs struct { unsigned q, r;}
; udiv_64by32_arm7m(unsigned d, unsigned long long n)
udiv_64by32_arm7m
    CMP     t, m                ;if (n >= (d << 32))
    BCS     overflow_32        ; goto overflow_32;
    RSB     m, m, #0           ;m = -d
    ADDS    q, r, r             ;{ [r,q] = 2*[r,q] - [d,0];
    ADCS    r, m, t, LSL#1      ; C= ([r,q] >= 0);}
    SUBCC   r, r, m             ;if (C==0) [r,q] += [d,0]
    GBLA    k                  ;the next 32 steps are the same
k      SETA    1                ;so we generate them using an
    WHILE    k<32              ;assembler while loop
        ADCS  q, q, q           ;{ [r,q] = 2*[r,q] + C - [d,0];
        ADCS  r, m, r, LSL#1     ;C= ([r,q] >= 0);}
        SUBCC r, r, m           ;if (C==0) [r,q] += [d,0]
k      SETA    k+1
    WEND
    ADCS    r0, q, q            ;insert final answer bit
    MOV     pc, lr              ;return { r0, r1 }
overflow_32
    MOV     r0, #-1
    MOV     r1, #-1
    MOV     pc, lr              ;return { -1, -1 }

```

这个程序的思想与 32 位/15 位除法类似。在第 k 次的试探减法后, 64 位值 $[r, q]$ 的高端 $(64-k)$ 位包含了余数, 低端 k 位包含了商的高 k 位。经过 32 次试探减法后, r 中存放了余数的值, q 中就保存了商的值。2 条 ADC 指令对 $[r, q]$ 左移一位, 把最后结果位插入到底部, 并从高 32 位减去除数。如果减法溢出, 则通过回加除数来纠正 r 的值。

7.3.2 无符号整数的 Newton-Raphson 除法

Newton-Raphson 迭代是一种求解数值方程的强有力技术。只要知道一个方程解的近似值, 通过迭代就可以快速逼近结果值。实际上, 这种逼近速度和运算结果的有效位数之间满足 2 次关系, 每次迭代运算可以使运算结果的有效位数增加大约 1 倍。Newton-Raphson 被广泛应用于计算高精度的倒数和平方根。这里将使用 Newton-Raphson 方法处理 16 位和 32 位整数和小数除法, 而且这种方法可以被运用到对任何位数的数值的除法运算中。

Newton-Raphson 技术适用于任何形式如 $f(x)=0$ 的方程式, 其中 $f(x)$ 是可微分的函数, 其导数是 $f'(x)$ 。首先假定方程式解 x 的一个近似值 x_n , 然后使用下面的迭代, 得到一个更精确的近似值 x_{n+1} 。

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (7.2)$$

图 7.3 证明了 Newton-Raphson 迭代求解 $f(x)=0.8-x^{-1}=0$ 的过程, 设 $x_0=1$ 作为初始近似值。经过 2 次迭代, 得到 $x_1=1.2$ 和 $x_2=1.248$, 此时已经非常逼近结果 1.25。

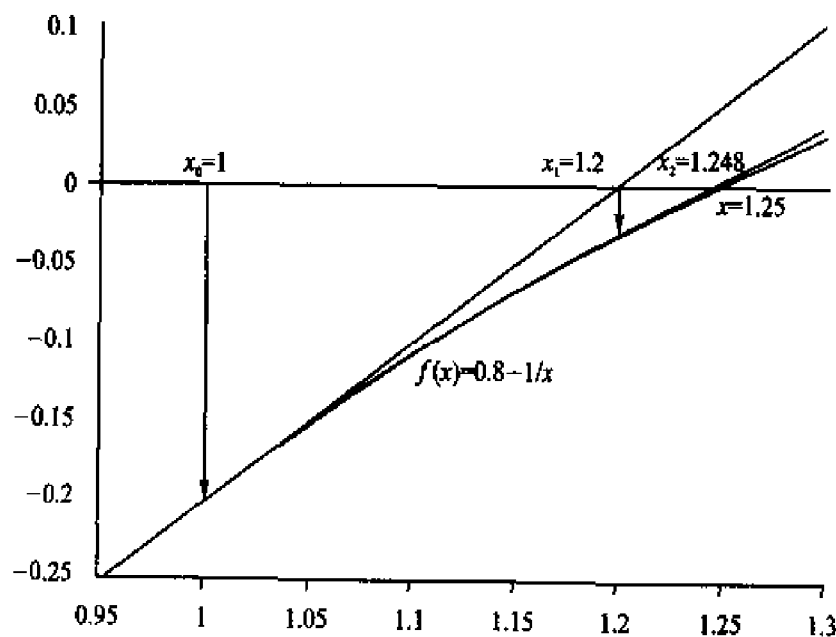


图 7.3 方程 $f(x)=0.8-1/x$ 的 Newton-Raphson 迭代

对于许多函数 f , 这种迭代可以快速逼近解 x 的值。从图形上看, x_{n+1} 的位置就是曲线在 x_n 的正切与 x 轴的相交点。

使用 Newton-Raphson 迭代方法, 可以仅使用整数乘法操作来计算 $2^N d^{-1}$ 。把 2^N 作为因子, 这样做对于估算 7.3.2.1 小节和 5.10.2 小节中用到的 $2^N d$ 是很有好处的。考虑下面的函数:

$$f(x) = d - \frac{2^N}{x} \quad (7.3)$$

方程 $f(x)=0$ 有解 $x=2^N d^{-1}$ 和导数 $f'(x)=2^N x^{-2}$ 。通过置换, Newton-Raphson 迭代为

$$x_{n+1} = x_n - \frac{d - 2^N x_n^{-1}}{2^N x_n^{-2}} = 2x_n - \frac{dx_n^2}{2^N} \quad (7.4)$$

可以看到, 这个迭代已经把上面的除法颠倒过来了, 原来的乘以 2^N , 除以 d , 变成了除以 2^N , 乘以 d 。下面 2 种情况特别有用:

- 当 $N=32$ 且 d 是整数时。在这种情况下, 可以快速估算出值 $2^N d^{-1}$, 然后用这个近似值来计算 n/d , 即两个无符号 32 位数的比。 $N=32$ 时的迭代见 7.3.2.1 小节。
- 当 $N=0$ 且 d 是用定点的形式表示的小数, 且满足 $0.5 \leq d < 1$ 时。在这种情况下, 使用迭代来计算 d^{-1} , 这对于计算一定范围内的定点形式的值 n 的 nd^{-1} 是很有帮助的。 $N=0$ 时的迭代见 7.3.3 小节。

通过 Newton-Raphson 实现的无符号 32 位/32 位除法

本小节给出了一个与 7.3.1.1 小节的程序不同的实现方法。下面的程序在 ARM9E 上充分利用了快速乘法器, 呈现出在最坏情况下的良好性能。使用整数 d 的 $N=32$ 时的 Newton-Raphson 迭代来估算整数 $2^{32}/d$; 然后把这个近似值与 n 相乘, 并除以 2^{32} , 得到一个商 $q=n/d$ 的近似值; 最后, 计算余数 $r=n-qd$, 并校正商和余数的舍入误差(rounding error)。

```

q      RN 0    ;input denominator d, output quotient q
r      RN 1    ;input numerator n, output remainder r
s      RN 2    ;scratch register
n      RN 3    ;scratch register
a      RN 12   ;scratch register

;_value_in_regs struct { unsigned q, r;}
;   udiv_32by32_arm9e(unsigned d, unsigned n)
udiv_32by32_arm9e ;instruction   number ; comment
CLZ     s, q                      ;01 ; find normalizing shift
MOVS    a, q, LSL s               ;02 ; perform a lookup on the
ADD     a, pc, a, LSR#25          ;03 ; most significant 7 bits

```

```

LDRNEB a, [a, #t32 - b32 - 64] ;04 : of divisor
b32    SUBS    s, s, #7          ;05 : correct shift
RSB     m, q, #0                ;06 :  $m = -d$ 
MOVPL   q, a, LSL s             ;07 :  $q \approx (1 \ll 32)/d$ 
;1st Newton iteration follows
MULPL   a, q, m                 ;08 :  $a = -q * d$ 
BMI     udiv_by_large_d         ;09 : large d trap
SMLAWT  q, q, a, q              ;10 :  $q \approx q - (q * q * d \gg 32)$ 
TEQ     m, m, ASR#1             ;11 : check for  $d = 0$  or  $d = 1$ 
;2nd Newton iteration follows
MULNE   a, q, m                 ;12 :  $a = -q * d$ 
MOVNE   s, #0                  ;13 :  $s = 0$ 
SMLALNE s, q, a, q             ;14 :  $q = q - (q * q * d \gg 32)$ 
BEQ     udiv_by_0_or_1          ;15 : trap  $d = 0$  or  $d = 1$ 
; q now accurate enough for a remainder  $r$ ,  $0 \leq r < 3 * d$ 
UMULL   s, q, r, q              ;16 :  $q = (r * q) \gg 32$ 
ADD     r, r, m                 ;17 :  $r = n - d$ 
MLA     r, q, m, r              ;18 :  $r = n - (q + 1) * d$ 
; since  $0 \leq n - q * d < 3 * d$ , thus  $-d \leq r < 2 * d$ 
CMN     r, m                    ;19 :  $t = r - d$ 
SUBCS   r, r, m                 ;20 : if  $(t < -d \parallel t \geq 0) r = r + d$ 
ADDC    q, q, #1                ;21 : if  $(-d \leq t \&\& t < 0) q = q + 1$ 
ADDEPL  r, r, m, LSL#1          ;22 : if  $(t \geq 0) \{ r = r - 2 * d$ 
ADDEPL  q, q, #2                ;23 :  $q = q + 2 \}$ 
BX      lr                      ;24 : return  $\{q, r\}$ 

```

udiv_by_large_d

; at this point we know $d \geq 2^{(31-6)} = 2^{25}$

```

SUB     a, a, #4                ;25 : set q to be an
RSB     s, s, #0                ;26 : underestimate of
MOV     q, a, LSR s             ;27 :  $(1 \ll 32)/d$ 
UMULL   s, q, r, q              ;28 :  $q = (n * q) \gg 32$ 
MLA     r, q, m, r              ;29 :  $r = n - q * d$ 
; q now accurate enough for a remainder  $r$ ,  $0 \leq r < 4 * d$ 
CMN     m, r, LSR#1             ;30 : if  $(r/2 \geq d)$ 
ADDCS   r, r, m, LSL#1          ;31 :  $\{ r = r - 2 * d;$ 
ADDCS   q, q, #2                ;32 :  $q = q + 2;$ 
CMN     m, r                    ;33 : if  $(r \geq d)$ 

```

```

        ADDCS    r, r, m           ;34 : { r = r - d;
        ADDCS    q, q, #1          ;35 :   q = q + 1; }
        BX      lr                 ;36 : return {q, r}

udiv_by_0_or_1
        ;carry set if d = 1, carry clear if d = 0
        MOVCS    q, r             ;37 : if (d == 1) { q = n;
        MOVCS    r, #0             ;38 :           r = 0; }
        MOVCC    q, #-1           ;39 : if (d == 0) { q = -1;
        MOVCC    r, #-1           ;40 :           r = -1; }
        BX      lr                 ;41 : return {q, r}

        ;table for 32 by 32 bit Newton Raphson divisions
        ;table[0] = 255
        ;table[i] = (1 << 14)/(64 + i) for i = 1,2,3,...,63
t32     DCB 0xff, 0xfc, 0xf8, 0xf4, 0xf0, 0xed, 0xea, 0xe6
        DCB 0xe3, 0xe0, 0xdd, 0xda, 0xd7, 0xd4, 0xd2, 0xcf
        DCB 0xcc, 0xca, 0xc7, 0xc5, 0xc3, 0xc0, 0xbe, 0xbc
        DCB 0xba, 0xb8, 0xb6, 0xb4, 0xb2, 0xb0, 0xae, 0xac
        DCB 0xaa, 0xa8, 0xa7, 0xa5, 0xa3, 0xa2, 0xa0, 0x9f
        DCB 0x9d, 0x9c, 0x9a, 0x99, 0x97, 0x96, 0x94, 0x93
        DCB 0x92, 0x90, 0x8f, 0x8e, 0x8d, 0x8c, 0x8a, 0x89
        DCB 0x88, 0x87, 0x86, 0x85, 0x84, 0x83, 0x82, 0x81

```

【证明 7.2】 代码的证明是相当棘手的。为了使证明和解释更简单明了，在每一行对各条指令都用行号作了标注。

注意：一些指令执行是有条件的，注释只给出了指令被执行时的情况。

根据除数 d 的大小，整个执行可以分成几段不同的代码流程，以分别处理不同的情况。在前面的代码中，使用 $1k$ 作为第 k 条指令的简短注释。

Case 1 $d=0$ ：包括返回语句，在 ARM9E 上共占用 27 个周期

对这种情况的验证如下：只有当 $q \neq 0$ 时，load 语句才会条件执行，从而避免了 I04 的查表。这就确保了不需要读完(read off)表格的开头。由于 I01 设置了 $s=32$ ，在 I09 就不会有任何跳转。I06 设置了 $m=0$ ，因此 I11 设置了 Z 标志，并清除了进位标志。在 I15 跳转到特定情况的代码。

Case 2 $d=1$ ：包括返回语句，在 ARM9E 上共占用 27 个周期

这种情况与 $d=0$ 的情况类似。尽管确实会发生在 I05 的表格查找，但结果可以忽略。

I06 设置了 $m = -1$, I11 设置了 Z 标志和进位标志。在 I37 的特定代码返回结果 $q = n, r = 0$ 。

Case 3 $2 \leq d < 2^{25}$: 包括返回语句, 在 ARM9E 上共占用 36 个周期

这种情况是最麻烦的。首先通过 d 的最高几位执行查表, 得到一个 $2^{32}/d$ 的估计值。I01 得到了满足 $2^{31} \leq d2^s < 2^{32}$ 的移位值 s 。I02 设置 $a = d2^s$, I03 和 I04 使用 a 的最高 7 位来执行表格查找, 这里用 i_0 表示, i_0 是介于 64 和 127 之间的一个索引值。把 d 截取 7 位引入一个误差值 f_0 :

$$i_0 = 2^{r-25}d - f_0 \quad 0 \leq f_0 < 1 \quad (7.5)$$

设置 a 的查找值 $a_0 = \text{table}[i_0 - 64] = 2^{14}i_0^{-1} - g_0$, g_0 是表的截断误差, $0 \leq g_0 \leq 1$ 。

然后

$$a_0 = \frac{2^{14}}{i_0} - g_0 = \frac{2^{14}}{i_0} \left(1 - \frac{g_0 i_0}{2^{14}}\right) = \frac{2^{14}}{i_0 + f_0} \left(1 + \frac{f_0}{i_0} - g_0 \frac{i_0 + f_0}{2^{14}}\right) \quad (7.6)$$

注意从式(7.5)得到 $i_0 + f_0 = 2^{s-25}d$, 并且各误差项并入 e_0 :

$$a_0 = \frac{2^{39-s}}{d} (1 - e_0) \quad e_0 = g_0 \frac{i_0 + f_0}{2^{14}} - \frac{f_0}{i_0} \quad (7.7)$$

因为通过 s 的选择, 得到 $64 \leq i_0 \leq i_0 + f_0 < 128$, 这样 $-f_0 2^{-6} \leq e_0 \leq g_0 2^{-7}$ 。由于 $d < 2^{25}$, 且 $s \geq 7$, I05 和 I07 计算下面的值放在寄存器 q 中:

$$q_0 = 2^{s-7}a_0 = \frac{2^{32}}{d}(1 - a_0) \quad (7.8)$$

这是 $2^{32}d^{-1}$ 的一个比较好的初始估计值。现在进行 2 次 Newton-Raphson 迭代, 以提高估计值的精确度。根据方程式(7.9), I08 和 I10 更新了寄存器 a 和 q 的值变成了 a_1 和 q_1 。I08 使用 $m = -d$ 来计算 a_1 的值。由于 $d \geq 2$, 当 $d = 2$ 时, 就有 $q_0 < 2^{31}$, 然后 $f_0 = 0, i_0 = 64, g_0 = 1, e_0 = 2^{-8}$ 。因此, 在 I10 就可以使用有符号乘累加指令 SMLAWT 来计算 q_1 。

$$\left. \begin{aligned} a_1 &= 2^{32} - dq_0 = 2^{32}e_0 \\ q_1 &= q_0 + (((a_1 \gg 16)q_0) \gg 16) \end{aligned} \right\} \quad (7.9)$$

右移分别引入了截断误差 $0 \leq f_1 < 1$ 和 $0 \leq g_1 < 1$ 。

$$q_1 = q_0 + (a_1 2^{-16} - f_1)q_0 2^{-16} - g_1 = \frac{2^{32}}{d}(1 - e_0^2 - f_1(1 - e_0)2^{-16}) - g_1 \quad (7.10)$$

$$q_1 = \frac{2^{32}}{d}(1 - e_1), \text{ 这里 } e_1 = e_0^2 + f_1(1 - e_0)2^{-16} + g_1 d 2^{-32} \quad (7.11)$$

这个新的近似值 q_1 更加精确, 其误差值 $e_1 \approx e_0^2$ 。I12, I13 和 I14 实现了第 2 次 Newton-Raphson 迭代, 并且更新了寄存器 a 和 q 的值为 a_2 和 q_2 。

$$a_2 = 2^{32} - dq_1 = 2^{32}e_1, \quad q_2 = q_1 + ((a_2 q_1) \gg 32) \quad (7.12)$$

ARM 嵌入式系统开发

再一次移位引入了截断误差 $0 \leq g_2 < 1$:

$$q_2 = q_1 + a_2 q_1 2^{-32} - g_2 = \frac{2^{32}}{d} (1 - e_2), \text{ 这里 } e_2 = e_1^2 + g_2 d 2^{-32} < e_1^2 + d 2^{-32} \quad (7.13)$$

现在的近似值 $2^{32} d^{-1}$ 已相当精确了。在方程式 (7.14), I16 通过设置 q 的值为 q_3 求估计值 n/d 。这个移位引入了一个舍入误差 $0 \leq g_3 < 1$:

$$q_3 = (nq_2) \gg 32 = nq_2 2^{-32} - g_3 = \frac{n}{d} - e_3, \text{ 这里 } e_3 = \frac{n}{d} e_2 + g_3 < \frac{n}{d} e_1^2 + 2 \quad (7.14)$$

误差 e_3 是一个比较小的正数, 但到底有多小呢? 通过证明 $e_1^2 < d 2^{-32}$, 可得到 $0 \leq e_3 < 3$ 。下面分几种情况来讨论。

Case 3.1 $2 \leq d \leq 16$

在这种情况下, 各截取值没有丢弃任何位, 因而 $f_0 = f_1 = g_1 = 0$, 进而有 $e_1 = e_0^2$, 且 $e_0 = i_0 g_0 2^{-14}$ 。表格 7.2 明确地计算了 i_0 和 g_0 的值。

表 7.2 对较小值 d 的误差值

d	i_0	g_0	e_0	$2^{32} e_1^2$
2, 4, 8, 16	64	1	2^{-8}	1
3, 6, 12	96	2//3	2^{-8}	1
5, 10	80	4//5	2^{-8}	1
7, 14	112	2//7	2^{-9}	<1
9	72	5//9	$5 * 2^{-11}$	<1
11	88	2//11	2^{-10}	<1
13	104	7//13	$7 * 2^{-11}$	<1
15	120	8//15	2^{-8}	<1

Case 3.2 $16 < d \leq 256$

则 $f_0 \leq 0.5$ 意味着 $|e_0| \leq 2^{-7}$ 。由于 $f_1 = g_1 = 0$, 就有 $e_1^2 \leq 2^{-28} < d 2^{-32}$ 。

Case 3.3 $256 < d < 512$

则 $f_0 \leq 1$ 意味着 $|e_0| \leq 2^{-6}$ 。由于 $f_1 = g_1 = 0$, 就有 $e_1^2 \leq 2^{-24} < d 2^{-32}$ 。

Case 3.4 $512 < d < 2^{25}$

则 $f_0 \leq 1$ 意味着 $|e_0| \leq 2^{-6}$ 。因此, $e_1 < 2^{-12} + 2^{-15} + d 2^{-32}$ 。设 $D = \sqrt{d 2^{-32}}$, 那么 $2^{-11.5} \leq D < 2^{-3.5}$ 。所以 $e_1 < D (2^{-0.5} + 2^{-3.5} + 2^{-3.5}) < D$, 就是所求的结果。

已知 $e_3 < 3$, I16~I23 计算了 3 个可能的 n/d 的结果 q_3, q_3+1, q_3+2 ; 然后计算余数 $r =$

$n-dq_3$, 并从余数减去 d , 加上 q , 直到 $0 \leq r < d$ 。

Case 4 $2^{25} \leq d$: 包括返回语句, 在 ARM9E 上共占用 32 个周期

这种情况开始与 Case 3 相同, 对 i_0 和 a_0 有相同的方程式。但随后跳转到 I25, 对 a_0 减去 4, 并右移 $(7-s)$ 位。这样就在式 (7.15) 中给出了近似值 q_0 。减去 4 的运算使得 q_0 是一个 $2^{32}d^{-1}$ 的低估值。对一些截断误差 $0 \leq g_0 < 1$:

$$q_0 = \left(\frac{2^{14}}{i_0} - 4 \right) \gg (7-s) = \frac{2^{s+7}}{i_0} - 2^{s-5} - g_0 \quad (7.15)$$

$$q_0 = \frac{2^{32}}{d} - e_0, \text{ 这里 } e_0 = 2^{s-5} + g_0 - \frac{2^{32}}{d} \frac{f_0}{i_0} \quad (7.16)$$

由于 $(2^{32}d^{-1})(f_0i_0^{-1}) \leq 2^{s-5}2^{-6} = 2^{s-6}$, 就得到 $0 \leq e_0 < 3$ 。I28 设置 q 的估计商值为 q_1 。对截断误差 $0 \leq g_1 < 1$:

$$q_1 = (nq_0) \gg 32 = \frac{nq_0}{2^{32}} - g_1 = \frac{n}{d} - \frac{n}{2^{32}}e_0 - g_1 \quad (7.17)$$

因此, q_1 是 n/d 的低估值, 误差不超过 4。最后几步 I28~I35, 使用一个 2 步的二分搜索 (binary search) 来确定最后的结果。

7.3.3 无符号小数 Newton-Raphson 除法

本小节将介绍利用 Newton-Raphson 技术来实现小数除法运算。小数的值是用定点算法表示的, 这对 DSP 应用是很有益的。

对于小数除法, 首先要按比例来缩放除数, 使其在范围 $0.5 \leq d < 1.0$ 之间; 然后使用查表来提供一个 d^{-1} 的近似值 x_0 ; 最后, 以 $N=0$ 进行 Newton-Raphson 迭代。从 7.3.2 小节可知, 这个迭代为:

$$x_{i+1} = 2x_i - dx_i^2 \quad (7.18)$$

随着 i 的增加, x_i 的精确度越来越高。为了得到最快速的实现, 在 i 值比较小时, 可以使用低精度的乘法, 然后随着每次迭代逐步提高精度。

这样就得到了比较短且快速的程序。7.3.3.3 小节给出了一个 15 位小数除法的程序, 7.3.3.4 小节给出了一个 31 位小数除法的程序。此外, 对程序进行了相当困难的证明, 也就是证明对所有可能的输入情况都可以得到正确的结果。对于 31 位除法, 不可能对每一个除数和被除数的组合都一一测试, 所以必须证明代码的正确性。7.3.3.1 小节和 7.3.3.2 小节涉及一些在 7.3.3.3 小节和 7.3.3.4 小节证明中需要的数学理论。如果读者对这些理论不感兴趣, 可以直接跳到 7.3.3.3 小节。

整个分析过程, 使用了下面的符号:

- d 是按比例缩放过的小数, $0.5 \leq d < 1$;

- i 是迭代的计数值;
- k_i 是 x_i 精确度的位数, 并保证 $k_{i+1} > k_i \geq 3$;
- x_i 是 d^{-1} 的一个 k_i 位的近似值, 满足 $0 \leq x_i \leq 2 - 2^{2-k_i}$;
- x_i 是 2^{1-k_i} 的倍数;
- $e_i = \frac{1}{d} - x_i$ 是 x_i 的误差值, 并保证 $|e_i| \leq 0.5$ 。

随着每一次迭代, 增加 k_i , 减小误差 e_i 。首先来看一下如何计算一个好的初始估计值 x_0 。

7.3.3.1 理论: Newton-Raphson 除法的初始估计值

如果对 Newton-Raphson 理论不感兴趣, 可以跳过这 2 小节, 直接跳到 7.3.3.3 小节。

首先通过一张按照 d 的最高有效位进行查找的表格来找到 d^{-1} 的初始估计值 x_0 。为了平衡表格大小和精确度之间的关系, 通过 d 的最前 8 个小数位作为索引值, 返回一个 9 位的估计值 x_0 。由于 d 和 x_0 的第一位都是 1, 那么就只需要一张包含 128 个 8 位入口的查找表格 (即该表格有 128 个表项)。

假定 a 是 d 的最高位及其后面 7 位的整数表示形式。 d 的范围为: $(128+a)2^{-8} \leq d < (129+a)2^{-8}$ 。取中间点 $c = (128.5+a)2^{-8}$, 定义查找表格为

$$\text{table}[a] = \text{round}(256.0/c) - 256;$$

这是浮点表示形式, round 表示取小数的最接近的整数值。在不支持浮点运算的情况下, 为了方便计算, 可以把它简化成一个整数的形式:

$$\text{table}[a] = (511 * (128 - a)) / (257 + 2 * a);$$

很明显, 表格的所有入口都在范围 0~255 之间。开始 Newton-Raphson 迭代之前, 设置 $x_0 = 1 + \text{table}[a]2^{-8}$, $k_0 = 9$ 。看一下 7.3.3.3 小节, 将会关注到下面的误差项:

$$E = d^2 e_0^2 + d 2^{-16} \quad (7.19)$$

首先, 看 $d|e_0|$, 如果 $e_0 \leq 0$, 那么

$$d|e_0| = x_0 d - 1 < x_0(129+a)2^{-8} - 1 \quad (7.20)$$

$$d|e_0| < ((256 + \text{table}[a])(129+a) - 2^{-16})2^{-16} \quad (7.21)$$

如果 $e_0 \geq 0$, 那么

$$d|e_0| = 1 - x_0 d \leq 1 - x_0(128+a)2^{-8} \quad (7.22)$$

$$d|e_0| < (2^{16} - (256 + \text{table}[a])(128+a))2^{-16} \quad (7.23)$$

尝试 a 的所有可能值, 就会发现 $d|e_0| < 299 \times 2^{-16}$ 。这是一个最好的可能范围。取 $d = (133-e)2^{-16}$, 正数 e 的值越小, 就能得到越接近的范围。用同样的方法也可以找到 E 的精确范围:

$$E2^{32} < ((256 + \text{table}[a])(128 + a) - 2^{16})^2 + (129 + a)2^8 \quad (e_0 \leq 0) \quad (7.24)$$

$$E2^{32} < (2^{16} - (256 + \text{table}[a])(128 + a))^2 + (129 + a)2^8 \quad (e_0 \geq 0) \quad (7.25)$$

尝试 a 的所有可能值,可以得到很精确的范围: $E < 2^{-11}$ 。因为表格的最大入口值为 254,这就验证了 $x_0 \leq 2 - 2^{-7}$ 。

7.3.3.2 理论: Newton-Raphson 小数迭代的精确度

本小节将分析每次小数 Newton-Raphson 迭代所引入的误差:

$$x_{i+1} = 2x_i - dx_i^2 \quad (7.26)$$

要精确计算这个迭代,通常是比较缓慢的。由于 x_i 最多只能精确到 k_i 位,所以计算超过 $2k_i$ 位精度是毫无意义的。下面给出了计算迭代的实用方法。迭代保存了在 7.3.3 小节中定义的 x_i 和 e_i 的界限。

① 正确计算 x_i^2

$$x_i^2 = \left(\frac{1}{d} - e_i\right)^2 \text{ 且 } 0 \leq x_i^2 \leq 4 - 2^{4-k_i} + 2^{4-2k_i} \quad (7.27)$$

② 计算 d 的低估计值 d_i ,通常 d 为 $2k_i$ 位。只需要

$$0.5 \leq d_i = d - f_i \text{ 和 } 0 \leq f_i \leq 2^{-4} \quad (7.28)$$

③ 计算 $d_i x_i^2$ 的 $k_{i+1} + 1$ 位的近似值 y_i ,且 $0 \leq y_i < 4$,确保 y_i 尽可能精确。然而,也需要使误差值 g_i 满足

$$y_i = d_i x_i^2 - g_i, -2^{-2} \leq g_i \leq 2^{3-2k_i} - 2^{2-k_{i+1}} \quad (7.29)$$

④ 使用一个正确的减法,计算新的估计值 $x_{i+1} = 2x_i - y_i$ 。将会证明 $0 \leq x_{i+1} < 2$,因此结果为 k_{i+1} 位。

新的 k_{i+1} 位的估计值 x_{i+1} 满足 7.3.3.1 小节中的特性,接下来计算新的误差 e_{i+1} 。首先检查 x_{i+1} 的范围:

$$x_{i+1} = 2x_i - d_i x_i^2 + g_i \leq 2x_i - 0.5x_i^2 + g_i \quad (7.30)$$

后面关于 x_i 的多项式对于 $x_i \leq 2$ 有正的斜率,这样就可以在 x_i 是最大值时,多项式也达到最大值。因此使用关于 g_i 的界限:

$$x_{i+1} \leq 2(2 - 2^{2-k_i}) - 0.5(4 - 2^{4-k_i} + 2^{4-2k_i}) + g_i \leq 2 - 2^{2-k_{i+1}} \quad (7.31)$$

另一方面,由于 $|e_i| \leq 0.5, g_i \geq -0.25$,就有

$$x_{i+1} \geq 2x_i - 1.5x_i + g_i \geq 0 \quad (7.32)$$

最后,计算新的误差:

$$e_{i+1} = \frac{1}{d} - x_{i+1} = \frac{1}{d} - 2x_i + (d - f_i)x_i^2 - g_i = de_i^2 - f_i x_i^2 - g_i \quad (7.33)$$

容易验证得到 $|e_{i+1}| \leq 0.5$ 。

7.3.3.3 使用 Newton-Raphson 的 Q15 定点除法

当计算比例 nd^{-1} 的一个 Q15 表示时, 其中 n 和 d 都是 16 位的正整数, 且 $0 \leq n < d < 2^{15}$ 。其实就是要计算

$$q = (n \ll 15) / d$$

使用 7.3.1.2 小节中的程序 `udiv_32by16_arm7m`, 通过试探减法可完成上述运算。然而, 下面的程序在 ARMV5E 上可占用更少的周期数, 而计算出同样的结果。如果只需要一个结果的近似值, 那么可以把 I15~I18 的指令移去, 这几条指令是用来纠正初始估计值的误差的。

这个程序在许多地方危险地转向似乎不正确, 所以随后需要证明其正确性。证明使用了 7.3.3.2 小节的理论。如果代码要求对另外的 ARM 核进行改变或优化, 那么这个证明可以作为一个有用的参考。在 ARM9E 上整个程序包括返回指令占用了 24 个周期。如果 $d \leq n < 2^{15}$, 那么就返回饱和值 `0x7fff`。

```

q      RN 0      ;input denominator d, quotient estimate q
r      RN 1      ;input numerator n, remainder r
s      RN 2      ;normalisation shift, scratch
d      RN 3      ;Q15 normalised denominator  $2^{14} \leq d < 2^{15}$ 

```

```

;unsigned udiv_q15_arm9e(unsigned d, unsigned q)

```

```

udiv_q15_arm9e ;instruction      number ; comment
    CLZ      s, q                ;01 ; choose a shift s to
    SUB      s, s, #17           ;02 ; normalize d to the
    MOVS     d, q, LSL s         ;03 ; range  $0.5 \leq d < 1$  at Q15
    ADD      q, pc, d, LSR #7    ;04 ; look up q, a Q8
    LDRNEB   q, [q, #t15 - b15 - 128] ;05 ; approximation to  $1/d$ 
b15    MOV     r, r, LSL s        ;06 ; normalize numerator
    ADD      q, q, #256          ;07 ; part of table lookup
    ;q is now a Q8, 9-bit estimate to  $1/d$ 
    SMULBB   s, q, q            ;08 ;  $s = q * q$  at Q16
    CMP      r, d               ;09 ; check for overflow
    MUL      s, d, s            ;10 ;  $s = q * q * d$  at Q31
    MOV      q, q, LSL #9       ;11 ; change q to Q17
    SBC      q, q, s, LSR #15    ;12 ;  $q = 2 * q - q * q * d$  at Q16

```

```

; q is now a Q16, 17-bit estimate to 1//d
SMULWB    q, q, r                ;13 : q approx n//d at Q15
BCS       overflow_15           ;14 : trap overflow case
SMULBB    s, q, d                ;15 : s = q * d at Q30
RSB       r, d, r, LSL #15      ;16 : r = n - d at Q30
CMP       r, s                  ;17 : if (r >= s)
ADDEPL    q, q, #1              ;18 : q++
BX        lr                    ;19 : return q

overflow_15
LDR       q, =0x7FFF            ;20 : q = 0x7FFF
BX        lr                    ;21 : return q

; table for fractional Newton Raphson division
; table[a] = (int)((511 * (128 - a)) / (257 + 2 * a)) 0 <= a < 128
t15       DCB 0xfe, 0xfa, 0xf6, 0xf2, 0xef, 0xeb, 0xe7, 0xe4
          DCB 0xe0, 0xdd, 0xd9, 0xd6, 0xd2, 0xcf, 0xcc, 0xc9
          DCB 0xc6, 0xc2, 0xbf, 0xbc, 0xb9, 0xb6, 0xb3, 0xb1
          DCB 0xae, 0xab, 0xa8, 0xa5, 0xa3, 0xa0, 0x9d, 0x9b
          DCB 0x98, 0x96, 0x93, 0x91, 0x8e, 0x8c, 0x8a, 0x87
          DCB 0x85, 0x83, 0x80, 0x7e, 0x7c, 0x7a, 0x78, 0x75
          DCB 0x73, 0x71, 0x6f, 0x6d, 0x6b, 0x69, 0x67, 0x65
          DCB 0x63, 0x61, 0x5f, 0x5e, 0x5c, 0x5a, 0x58, 0x56
          DCB 0x54, 0x53, 0x51, 0x4f, 0x4e, 0x4c, 0x4a, 0x49
          DCB 0x47, 0x45, 0x44, 0x42, 0x40, 0x3f, 0x3d, 0x3c
          DCB 0x3a, 0x39, 0x37, 0x36, 0x34, 0x33, 0x32, 0x30
          DCB 0x2f, 0x2d, 0x2c, 0x2b, 0x29, 0x28, 0x27, 0x25
          DCB 0x24, 0x23, 0x21, 0x20, 0x1f, 0x1e, 0x1c, 0x1b
          DCB 0x1a, 0x19, 0x17, 0x16, 0x15, 0x14, 0x13, 0x12
          DCB 0x10, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09
          DCB 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01

```

【证明 7.3】 程序用指令 I01, I02, I03, I06 规格化 d 和 n , 使得 $2^{14} \leq d < 2^{15}$ 。由于会对除数和被除数进行相同数目的左移还原, 所以这不会影响计算的结果。考虑到 d 是 Q15 形式的定点小数, 且 $0.5 \leq d < 1$ 。I09 和 I14 是溢出陷阱, 能捕获 $n \geq d$ 的情况, 也包含了 $d=0$ 的情况。假定从现在开始没有溢出, I04, I05, I07 设置 q 为 7.3.1.1 小节中描述的 d^{-1} 的 9 位 Q8 形式的初始估计值 x_0 。由于 d 满足 $0.5 \leq d < 1$, 在查找表时减去 128, 这样 $d=0.5$ 就与查找表的第一个入口相对应了。

接下来执行一次 Newton-Raphson 迭代。I08 设置 a 为 x 的 Q16 形式的平方, I10 设置 a 为 dx_0^2 的 Q31 形式的值。这里需要留意一个细节: 验证这个值不会溢出一个无符号 Q31 的表示范围, 实际上:

$$dx_0^2 = \frac{x_0^2}{x_0 + e_0} = \left(1 + x_0 - \frac{e_0}{x_0 + e_0}\right) \quad (7.34)$$

当 x_0 尽可能大, 而 e_0 尽可能负时, 上面的等式可以得到最大值。这时 $d = 0.5 + 2^{-8} - 2^{-15}$, $x_0 = 2 - 2^{-7}$, $dx_0^2 < 2 - 2^{-13} < 2$ 。

最后, I11 和 I12 设置 q 为新的 Q16 近似值 x_1 。由于进位标志在 I09 清除了, SBC 指令低估倒数。

$$x_1 = 2x_0 - dx_0^2 + g_0, \text{ 这里 } -2^{-16} \leq g_0 < 0 \quad (7.35)$$

使用方程式 (7.33) 来求出新的误差:

$$0 \leq e_1 = de_0^2 - g_0 \leq de_0^2 + 2^{-16} \quad (7.36)$$

I13 计算商 nd^{-1} 的 Q15 近似值 q_1 :

$$q_1 = nx_1 - h_1 = \frac{n}{d} - e_2 \quad (7.37)$$

这里 $0 \leq h_1 < 2^{-15}$ 是截断误差, 并且

$$e_2 = ne_1 + h_1 < de_1 + h_1 < E + h_1 < 2^{-14} \quad (7.38)$$

E 的范围可以从 7.3.3.1 小节得到。因此, q_1 是对 nd^{-1} 的误差小于 2^{-14} 的一个低估值。最后, I15, I16, I17 和 I18 计算余数 $n - qd$, 并校正近似值到 Q15 精度。

7.3.3.4 使用 Newton-Raphson 的 Q31 定点除法

这里要计算一个比率 nd^{-1} 的 Q31 表示, 其中 n 和 d 都是 32 位的正整数, 且满足 $0 \leq n < d < 2^{31}$ 。其实就是要计算

$$q = (\text{unsigned int})(((\text{unsigned long long}) n \ll 31) / d);$$

可以使用 7.3.1.2 小节的程序 `udiv_64by32_arm7m`, 用试探减法来完成这种除法。然而, 下面的程序在 ARM9E 上只需要更少的周期数, 就可以正确地计算出相同的结果。如果只需要一个结果的近似值, 那么可以把 I21~I29 的指令去掉, 这几条指令是用来校正初始估计值误差的。

如同前面的小节, 汇编代码后面还带了正确性的证明。这个程序在 ARM9E 上, 包括返回指令共占用了 46 个周期。程序使用了与 7.3.3.3 小节中的 Q15 除法程序同样的查找表格。

```
q      RN 0      ;input denominator d, quotient estimate q
r      RN 1      ;input numerator n, remainder high r
```


s RN 2 ;normalisation shift, scratch register
d RN 3 ;Q31 normalised denominator $2^{-30} < d < 2^{-31}$
a RN 12 ;scratch

;unsigned udiv_q31_arm9e(unsigned d, unsigned q)

```

udiv_q31_arm9e ;      instruction number ; comment
    CLZ      s, q                ;01 ; choose a shift s to
    CMP      r, q                ;02 ; normalize d to the
    MOVCC    d, q, LSL s        ;03 ; range  $0.5 < d < 1$  at Q32
    ADDCC    q, pc, d, LSR # 24 ;04 ; look up q, a Q8
    LDRCCB   q, [q, #t15 - b31 - 128] ;05 ; approximation to  $1//d$ 
b31  MOVCC    r, r, LSL s        ;06 ; normalize numerator
    ADDCC    q, q, #256          ;07 ; part of table lookup
    ;q is now a Q8, 9-bit estimate to  $1//d$ 
    SMULBBCC a, q, q            ;08 ;  $a = q * q$  at Q16
    MOVCS    q, #0x7FFFFFFF     ;09 ; overflow case
    UMULLCC  s, a, d, a         ;10 ;  $a = q * q * d$  at Q16
    BXCS     lr                 ;11 ; exit on overflow
    RSB      q, a, q, LSL # 9    ;12 ;  $q = 2 * q - q * q * d$  at Q16
    ;q is now a Q16, 17-bit estimate to  $1//d$ 
    UMULL    a, s, q, q         ;13 ;  $[s, a] = q * q$  at Q32
    MOVS     a, a, LSR # 1       ;14 ; now halve  $[s, a]$  and
    ADC      a, a, s, LSL # 31   ;15 ; round so  $[N, a] = q * q$  at
    MOVS     s, s, LSL # 30      ;16 ; Q31, C = 0
    UMULL    s, a, d, a         ;17 ;  $a = a * d$  at Q31
    ADDMI    a, a, d            ;18 ; if (N)  $a += 2 * d$  at Q31
    RSC      q, a, q, LSL # 16   ;19 ;  $q = 2 * q - q * q * d$  at Q31
    ;q is now a Q31 estimate to  $1/d$ 
    UMULL    s, q, r, q         ;20 ; q approx  $n//d$  at Q31
    ;q is now a Q31 estimate to num/den, remainder  $< 3 * d$ 
    UMULL    s, a, d, q         ;21 ;  $[a, s] = d * q$  at Q62
    RSBS     s, s, #0           ;22 ;  $[r, s] = n - d * q$ 
    RSC      r, a, r, LSR # 1    ;23 ; at Q62
    ; $[r, s] = (r << 32) + s$  is now the positive remainder  $< 3 * d$ 
    SUBS     s, s, d            ;24 ;  $[r, s] = n - (d + 1) * q$ 
    SBCS     r, r, #0           ;25 ; at Q62
    ADDEPL   q, q, #1           ;26 ; if ( $[r, s] >= 0$ )  $q++$ 
    SUBS     s, s, d            ;27 ;  $[r, s] = [r, s] - d$ 

```

```

SBCS      r, r, #0           ;28 : at Q62
ADDPL     q, q, #1           ;29 : if ([r,s]>=0) q++
BX        lr                 ;30 : return q

```

【证明 7.4】 首先检查 $n < d$ 是否成立, 如果不成立, 则经过一系列的条件跳转指令在 I11 返回饱和值 $0x7fffff$; 否则 d 和 n 都规格化为 Q31 形式, 且 $2^{30} \leq d < 2^{31}$ 。I07 与 7.3.3.3 小节中的一样, 设置 q 为 Q8 形式的初始近似值 x_0 。

I08, I10 和 I12 执行第一次 Newton-Raphson 迭代。I08 设置 a 为 x_0^2 的 Q16 表示。I10 设置 a 为 $dx_0^2 - g_0$ 的 Q16 表示, 且误差的范围为 $0 \leq g_0 < 2^{-16}$ 。I12 设置 x 为 x_1 的 Q16 表示, 即 d^{-1} 的新的近似值。方程式 (7.33) 给出了这个近似值的误差 $e_1 = de_0^2 - g_0$ 。

I13~I19 执行了第 2 次 Newton-Raphson 迭代。I13~I15 设置 a 为 $a_1 = x_1^2 + b_1$ 的 Q31 表示, b_1 是误差。由于在 I15 使用 ADC 指令, 计算就比较集中 (round up), 因此 $0 \leq b_1 \leq 2^{-32}$ 。由于 $2^{33} - 1$ 和 $2^{34} - 1$ 不是平方数, 所以 ADC 指令不会溢出。但是, a_1 可能溢出一个 Q31 表示的范围。如果溢出发生, 则 I16 清除进位标志, 记录 N 标志, 这样 $a_1 \geq 2$ 。I17 和 I18 设置 a 为 $y_1 = da_1 - c_1$ 的 Q31 表示, 舍入误差 $0 \leq c_1 < 2^{-31}$ 。由于进位标志清除了, I19 设置 q 为新的低估值的 Q31 表示:

$$x_2 = 2x_1 - d(x_1^2 + b_1) + c_1 - 2^{-31} = \frac{1}{d} - e_2 \quad (7.39)$$

$$e_2 = de_1^2 - c_1 + 2^{-31} + db_1 < de_1^2 + d2^{-32} + 2^{-31} \quad (7.40)$$

I20 设置 q 为商 $q_2 = nx_2 - b_2$ 的 Q31 表示, 舍入误差 $0 \leq b_2 < 2^{-31}$ 因此:

$$q_2 = \frac{n}{d} - e_3, \text{ 这里 } e_3 = ne_2 + b_2 < d^2 e_1^2 + d^2 2^{-32} + d2^{-31} + 2^{-31} \quad (7.41)$$

如果 $e_1 \geq 0$, 那么 $d^2 e_1^2 \leq d^4 e_0^2 < (2^{-15} - d2^{-16})^2$, 使用 7.3.3.1 小节中 E 的范围:

$$e_3 < 2^{-30} - d2^{-31} + d^2 2^{-32} + 2^{-31} < 3 \times 2^{-31} \quad (7.42)$$

如果 $e_1 < 0$, 那么 $d^2 e_1^2 \leq d^2 g_0^2 < 2^{-32}$ 。同样, $e_3 < 3 \times 2^{-31}$ 。在其它情况, q 是商的一个低估值, 误差不超过 3×2^{-31} 。I21~I23 计算余数, I24~I29 执行 2 个条件减法指令来校正 Q31 结果 q 。

7.3.4 有符号数除法

前面讨论了无符号数除法的实现。如果要对有符号数进行除法运算, 那么就要先把这些数变成无符号数, 再对结果加上相应的符号位。如果除数和被除数的符号位不一致, 那么

商是负数。余数的符号与被除数的符号一致。下面的例子显示了如何把有符号整数除法变成无符号除法,以及如何计算商和余数的符号位。

```

d      RN 0      ;input denominator, output quotient
r      RN 1      ;input numerator n, output remainder
sign   RN 12

;_value_in_regs struct { signed q, r;}
;   udiv_32by32_arm7m(signed d, signed n)
sdiv_32by32_arm7m
    STMFD    sp!, {lr}
    ANDS     sign, d, #1 << 31          ;sign = (d<0 ? 1 << 31 : 0);
    RSEMI    d, d, #0                   ;if (d<0) d = -d;
    EORS     sign, sign, r, ASR#32      ;if (r<0) sign = ~sign;
    RSBCS    r, r, #0                   ;if (r<0) r = -r;
    BL       udiv_32by32_arm7m          ;(d,r) = (r/d, r%d)
    MOVS     sign, sign, LSL#1          ;C = sign[31], N = sign[30]
    RSBCS    d, d, #0                   ;if (sign[31]) d = -d;
    RSEMI    r, r, #0                   ;if (sign[30]) r = -r;
    LDMFD    sp!, {pc}

```

注意:上面的程序与程序 `udiv_32by32_arm7m` 一样,使用 `r12` 来保存商和余数的符号位(见 7.3.1.1 小节)。

229

7.4 平方根

对于平方根,可使用与除法相同的技术来处理。可选择试探减法或 Newton-Raphson 迭代来实现。试探减法适合于结果不超过 16 位、精度要求比较低的情况;而对结果精度要求比较高的情况,可以使用 Newton-Raphson 迭代。7.4.1 小节和 7.4.2 小节分别介绍了试探减法和 Newton-Raphson 迭代的实现。

7.4.1 通过试探减法计算平方根

计算一个 32 位的无符号整数 d 的平方根,结果是 16 位无符号整数 q 和一个 17 位的无符号余数 r ,即

$$d = q^2 + r, 0 \leq r < 2q \quad (7.43)$$

ARM 嵌入式系统开发

起始时,设置 $q=0$ 和 $r=d$ 。接下来从最高可能位(第 15 位)开始,依次向下试探性地设置 q 的各个位。如果新的余数是正的,则设置该位。特别地,如果通过对 q 加上 2^n 来设置位 n ,那么新的余数是

$$r_{\text{new}} = d - (q + 2^n)^2 = (d - q^2) - 2^{n+1}q - 2^{2n} = r_{\text{old}} - 2^n(2q + 2^n) \quad (7.44)$$

因此,为了计算新的余数,可以尝试减去值 $2^n(2q + 2^n)$ 。如果减法得到一个非负的结果,那么就设置 q 的位 n 。下面的 C 代码算法实现了计算 $2n$ 位的输入值 d 的 n 位平方根 q :

```
unsigned usqr_simple(unsigned d, unsigned N)
{
    unsigned t, q = 0, r = d;

    do
    {
        /* calculate next quotient bit */
        N--;
        /* move down to next bit */
        t = 2 * q + (1 << N); /* new r = old r - (t << N) */
        if ((r >> N) >= t) /* if (r >= (t << N)) */
        {
            r -= (t << N); /* update remainder */
            q += (1 << N); /* update root */
        }
    } while (N);

    return q;
}
```

使用下面优化过的汇编程序来实现上面的算法,包括返回指令在内只需 50 个周期。这里巧妙之处在于计算结果的位 N 之前,寄存器 q 保存了值 $(1 << 30) | (q >> (N+1))$ 。如果对这个值进行循环左移 $(2N+2)$ 位,或者等价地右移 $(30-2N)$ 位,那么就有值 $t << N$,可以用于试探减法。

```
q      RN 0      ;input value, current square root estimate
r      RN 1      ;the current remainder
c      RN 2      ;scratch register

usqr_32 ;unsigned usqr_32(unsigned q)
        SUBS    r, q, #1 << 30      ;is q >= (1 << 15)^2?
        ADDCC   r, r, #1 << 30      ;if not restore
```

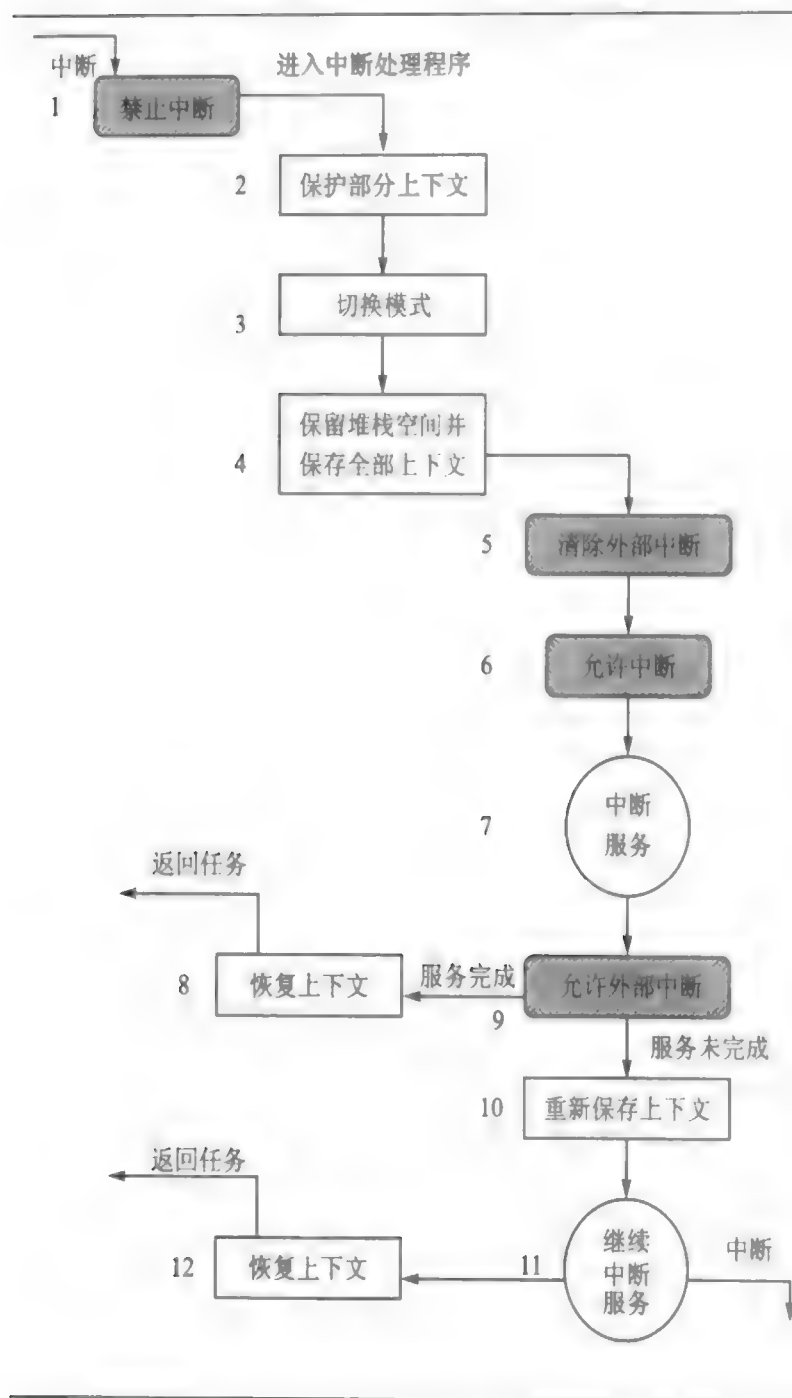


图 9.10 可重入中断处理

没有被优先级排序,则系统延迟性能会下降到与嵌套中断处理相同的水平。因为那样的话,低优先级中断能够抢占正在被服务的高优先级中断;同时,这也会导致在低优先级中断服务期间,高优先级中断得不到响应。

【例 9.10】 假设寄存器 `r13_irq` 已被设置为指向一个 12 字节的数据结构体,而不是指向标准的 IRQ 堆栈。`IRQ_spsr` 等偏移量用来指向这个结构的数据项。像所有的中断处理程序一样,需要做一些标准的定义,以修改寄存器 `cpsr` 和 `spsr`。

```

a      RN 12      ;scratch register/accumulator

rsqr_32 ;unsigned rsqr_32(unsigned q)
    CLZ      s, q                ;choose shift s which is
    BIC      s, s, #1           ;even such that  $d = (q \ll s)$ 
    MOVS     d, q, LSL s        ;is  $0.25 \leq d < 1$  at Q32
    ADDNE    q, pc, d, LSR #25  ;table lookup on top 7 bits
    LDRNEB   q, [q, #tab - base - 32] ;of d in range 32 to 127
base    BEQ    div_by_zero      ;divide by zero trap
    ADD      q, q, #0x100        ;table stores only bottom 8 bits
    ;q is now a Q8, 9-bit estimate to  $1/\sqrt{d}$ 
    SMULBB   a, q, q            ;a =  $q * q$  at Q16
    MOV      b, d, LSR #17      ;b = d at Q15
    SMULWB   a, a, b            ;a =  $d * q * q$  at Q15
    MOV      b, q, LSL #7       ;b = q at Q15
    RSB      a, a, #3 << 15     ;a =  $(3 - d * q * q)$  at Q15
    MUL      q, a, b            ;q =  $q * (3 - d * q * q)/2$  at Q31
    ;q is now a Q31 estimate to  $1/\sqrt{d}$ 
    UMULL    b, a, d, q         ;a =  $d * q$  at Q31
    MOV      s, s, LSR #1       ;square root halves the shift
    UMULL    b, a, q, a         ;a =  $d * q * q$  at Q30
    RSB      s, s, #15          ;reciprocal inverts the shift
    RSB      a, a, #3 << 30     ;a =  $(3 - d * q * q)$  at Q30
    UMULL    b, q, a, q         ;q =  $q * (3 - d * q * q)/2$  at Q31
    ;q is now a good Q31 estimate to  $1/\sqrt{d}$ 
    MOV      q, q, LSR s        ;undo the normalization shift
    BX      lr                  ;return q
div_by_zero
    MOV      q, #0xFFFFFFFF     ;maxium positive answer
    BX      lr                  ;return q

tab      ;tab[k] = round(256.0/sqrt((k + 32.3)/128.0)) - 256
DCB 0xfe, 0xf6, 0xef, 0xe7, 0xe1, 0xda, 0xd4, 0xce
DCB 0xc8, 0xc3, 0xbd, 0xb8, 0xb3, 0xae, 0xaa, 0xa5
DCB 0xa1, 0x9c, 0x98, 0x94, 0x90, 0x8d, 0x89, 0x85
DCB 0x82, 0x7f, 0x7b, 0x78, 0x75, 0x72, 0x6f, 0x6c
DCB 0x69, 0x66, 0x64, 0x61, 0x5e, 0x5c, 0x59, 0x57
DCB 0x55, 0x52, 0x50, 0x4e, 0x4c, 0x49, 0x47, 0x45

```

```

DCB 0x43, 0x41, 0x3f, 0x3d, 0x3b, 0x3a, 0x38, 0x36
DCB 0x34, 0x32, 0x31, 0x2f, 0x2d, 0x2c, 0x2a, 0x29
DCB 0x27, 0x26, 0x24, 0x23, 0x21, 0x20, 0x1e, 0x1d
DCB 0x1c, 0x1a, 0x19, 0x18, 0x16, 0x15, 0x14, 0x13
DCB 0x11, 0x10, 0x0f, 0x0e, 0x0d, 0x0b, 0x0a, 0x09
DCB 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01

```

类似地,为了计算 $d^{-1/k}$,可以对方程式 $f(x)=d-x^{-k}=0$ 使用 Newton-Raphson 迭代。

7.5 超越函数:log,exp,sin,cos

可以通过查表和级数展开来计算超越函数。这里,将讨论 4 个最常用的超越函数的实现方法:log,exp,sin 和 cos。在 DSP 应用中,为实现线性和对数形式的转换,需要对数和求幂函数。三角函数 sin 和 cos 在 2D 和 3D 图形及映射计算时是很有用的。

本节所有例子产生的结果都精确到 32 位,对很多应用来说,这个结果应该是足够了。也可以通过缩减级数展开来改善系统的性能;然而,这会损失一些精度。

7.5.1 以 2 为底的对数运算

假定 n 是 32 位的整数,求它的以 2 为底的对数 $s = \text{lb } n$,也就是 $n=2^s$ 。由于 s 满足 $0 \leq s < 32$,实际上是要找到一个 Q26 形式的对数 q ,使其满足 $q=s2^{26}$ 。使用 CLZ 或者 7.2 节中的替换方法,可以很容易计算出 s 的整数部分。这样就把数值范围缩小到了 $1 \leq n < 2$ 。首先对 n 的近似值 a 查表,找到 $\text{lb } a$ 和 a^{-1} 。由于

$$\text{lb } n = \text{lb } a + \text{lb } \frac{n}{a} \quad (7.47)$$

式中:lb 表示以 2 为底的对数。现在已经把问题简化为寻找 $\text{lb } (na^{-1})$ 。由于 na^{-1} 与 1 比较接近,可以使用 $\text{lb } (1+x)$ 的级数展开来提高结果的精确度:

$$\text{lb } (1+x) = \frac{\ln (1+x)}{\ln 2} = \frac{1}{\ln 2} \left(x - \frac{x^2}{2} + \frac{x^3}{3} - \dots \right) \quad (7.48)$$

式中:ln 表示以 e 为底的自然对数。可以把对数运算归结为 3 个步骤,见图 7.4:

- ① 使用 CLZ 指令来找到结果的位[31:26];
- ② 通过前 5 个小数位查表得到一个近似值;
- ③ 使用级数展开来计算,使近似值更精确。

下面的代码在 ARM9E 处理器上,包括返回指令共需要 31 个周期。结果的误差不超过 2^{-26} 。

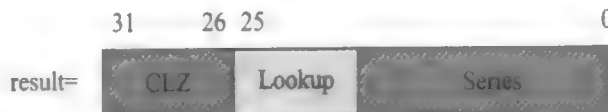


图 7.4 对数计算的 3 个步骤

```

n    RN 0    ;Q0 input, Q26 log2 estimate
d    RN 1    ;normalise input Q32
r    RN 2
q    RN 3
t    RN 12

```

```

;int ulog2_32(unsigned n)

```

```

ulog2_32

```

```

    CLZ    r, n
    MOV    d, n, LSL#1
    MOV    d, d, LSL r          ;1<= d<2 at Q32
    RSB    n, r, #31           ;integer part of the log
    MOV    r, d, LSR#27        ;estimate a = 1 + (r + 0.5)/32
    ADR    t, ulog2_table
    LDR    r, [t, r, LSL#3]!    ;r = lb a at Q26
    LDR    q, [t, #4]          ;q = 1/a at Q32
    MOV    t, #0
    UMLAL  t, r, d, r          ;r = (d/a) - 1 at Q32
    LDR    t, = 0x55555555     ;round(2^32/3)
    ADD    n, q, n, LSL#26     ;n + lb a at Q26
    SMULL  t, q, r, t          ;q = r/3 at Q32
    LDR    d, = 0x05c551d9     ;round(2^26/ln(2))
    SMULL  t, q, r, q          ;q = r^2/3 at Q32
    MOV    t, #0
    SUB    q, q, r, ASR#1      ;q = -r/2 + r^2/3 at Q32
    SMLAL  t, r, q, r          ;r = r^2/2 + r^3/3 at Q32
    MOV    t, #0
    SMLAL  t, n, d, r          ;n += r/lb 2 at Q26
    MOV    pc, lr

```

```

ulog2_table

```

```

;table[2 * i] = round(2^32/a)   where a = 1 + (i + 0.5)/32

```



```

,table[2 * i + 1] = round(2^26 * log2(a)) and 0 <= i < 32
DCD 0xfc0fc0fc, 0x0016e797, 0xf4898d60, 0x0043ace2
DCD 0xed7303b6, 0x006f2109, 0xe6c2b448, 0x0099574f
DCD 0xe070381c, 0x00c2615f, 0xda740da7, 0x00ea4f72
DCD 0xd4c77b03, 0x0111307e, 0xcf6474a9, 0x0137124d
DCD 0xca4587e7, 0x015c01a4, 0xc565c87b, 0x01800a56
DCD 0xc0c0c0c1, 0x01a33761, 0xbc52640c, 0x01c592fb
DCD 0xb81702e0, 0x01e726aa, 0xb40b40b4, 0x0207fb51
DCD 0xb02c0b03, 0x0228193f, 0xac769184, 0x0247883b
DCD 0xa8e83f57, 0x02664f8d, 0xa57eb503, 0x02847610
DCD 0xa237c32b, 0x02a20231, 0x9f1165e7, 0x02bef9ff
DCD 0x9c09c09c, 0x02db632d, 0x991f1a51, 0x02f7431f
DCD 0x964fda6c, 0x03129ee9, 0x939a85c4, 0x032d7b5a
DCD 0x90fdb0c09, 0x0347dcfe, 0x8e78356d, 0x0361c825
DCD 0x8c08c08c, 0x037b40e4, 0x89ae408a, 0x03944b1c
DCD 0x8767ab5f, 0x03acea7c, 0x85340853, 0x03c52286
DCD 0x83126e98, 0x03dcf68e, 0x81020408, 0x03f469c2

```

7.5.2 2 的乘幂

这是 7.5.1 小节反向操作。给出一个 Q26 表示的 $0 \leq x < 32$, 计算 2 的 x 次幂 2^x 。开始时, 把 x 分离成整数部分 n 和小数部分 d , 这样 $2^x = 2^d * 2^n$ 。要计算 2^d , 首先应找到 d 的近似值 a , 再寻找 2^a 。现在

$$2^d = 2^a \times \exp((d - a) \ln 2) \quad (7.49)$$

计算 $x = (d - a) \ln 2$, 并对 $\exp(x)$ 进行级数展开, 以提高精度:

$$\exp x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \dots \quad (7.50)$$

下面的汇编代码实现了上面的算法。结果的误差不超过 4。在 ARM9E 上整个程序包括返回指令, 共占用 31 个周期。

```

n      RN 0      ;input, integer part
d      RN 1      ;fractional part
r      RN 2
q      RN 3
t      RN 12

```

```

;unsigned uexp2_32(int n)
uexp2_32

```

```

MOV    d, n, LSL#6           ;d = fractional part at Q32
MOV    q, d, LSR#27          ;estimate a = (q + 0.5)/32
LDR    r, = 0xb17217f8       ;round(2^32 * lb 2)
BIC    d, d, q, LSL#27       ;d = d - (q/32) at Q32
UMULL   t, d, r, d            ;d = d * lb 2 at Q32
LDR    t, = 0x55555555       ;round(2^32/3)
SUB    d, d, r, LSR#6        ;d = d - lb 2/64 at Q32
SMULL   t, r, d, t           ;r = d/3 at Q32
MOVS    n, n, ASR#26         ;n = integer part of exponent
SMULL   t, r, d, r           ;r = d^2/3 at Q32
BMI     negative             ;catch negative exponent
ADD    r, r, d               ;r = d + d^2/3
SMULL   t, r, d, r           ;r = d^2 + d^3/3
ADR    t, uexp2_table
LDR    q, [t, q, LSL#2]      ;q = exp 2(a) at Q31
ADDS    r, d, r, ASR#1       ;r = d + d^2/2 + d^3/6 at Q32
UMULL   t, r, q, r           ;r = exp 2(a) * r at Q31 if r<0
RSB     n, n, #31            ;31 - (integer part of exponent)
ADDFP   r, r, q              ;correct if r>0
MOV     n, r, LSR n          ;result at Q0
MOV     pc, lr

negative
MOV     r0, #0               ;2^(-ve) = 0
MOV     pc, lr

```

uexp2_table

```

;table[i] = round(2^31 * exp2(a)) where a = (i + 0.5)/32
DCD 0xb164d1f4, 0x843a28c4, 0xb71f6197, 0xb8a14d575
DCD 0xb8d1adf5b, 0x9031dc43, 0x935a2b2f, 0x96942d37
DCD 0x99e04593, 0x9d3ed9a7, 0xa0b05110, 0xa43515ae
DCD 0xa7cd93b5, 0xab7a39b6, 0xaf3b78ad, 0xb311c413
DCD 0xb6fd91e3, 0xbaff5ab2, 0xbf1799b6, 0xc346ccda
DCD 0xc78d74c9, 0xcbec14ff, 0xd06333db, 0xd4f35aac
DCD 0xd99d15c2, 0xde60f482, 0xe33f8973, 0xe8396a50
DCD 0xed4f301f, 0xf281773c, 0xf7d0df73, 0xfd3e0c0d

```

7.5.3 三角函数

对于精度要求不是很高的应用(如产生正弦波和其它音频信号或一般图形处理),可以

使用查表来完成三角函数运算;然而还有许多对精度要求较高的应用,例如高精度的图形处理或全球定位。这里介绍的程序,可以实现精确到 32 位的 \sin 和 \cos 运算。

标准 C 库中的 \sin 和 \cos 函数使用弧度。弧度对于处理优化过的定点函数是比较麻烦的。首先,应对任何角度都要进行一次模 2π 运算;其次,还要判断角度在哪个象限。这里用模 2^{32} 运算来代替模 2π 运算,在任何处理器上它都是一个非常容易的操作。

现在先来定义新的基于 2 的三角函数 s 和 c 。这里角度被指定了一个比例, 2^{32} 表示一次旋转 (2π rad 或者 360°)。使用标准的模整数加法来增加角度值:

$$s(x) = \sin(2\pi x 2^{-32}) = \sin(\pi x 2^{-31}), c(x) = \cos(\pi x 2^{-31}) \quad (7.51)$$

在这种形式下, x 是旋转比例的 Q32 角度表示。 x 的最高 2 位表示角度在圆的哪个象限。首先使用 x 的最高 3 位,使 $s(x)$ 或 $c(x)$ 转化为 $0 \sim 1/8$ 圆的正弦或余弦;然后选择一个 x 的近似值 a ,使用查表来得到 $s(a)$ 和 $c(a)$ 。下面的公式把问题简化为寻找小角度的 \sin 和 \cos :

$$s(x) = s(a)\cos(\pi(x-a)2^{-31}) + c(a)\sin(\pi(x-a)2^{-31}) \quad (7.52)$$

$$c(x) = c(a)\cos(\pi(x-a)2^{-31}) + s(a)\sin(\pi(x-a)2^{-31}) \quad (7.53)$$

接下来,用弧度来计算小角度 $n = \pi(x-a)2^{-31}$,并使用下面的级数展开来进一步提高精度:

$$\sin x = x - \frac{x^3}{6} + \dots, \cos x = 1 - \frac{x^2}{2} + \dots \quad (7.54)$$

可以使用下面的汇编代码来实现上面 \sin 和 \cos 的算法。结果用 Q30 形式表示,误差不得超过 4×2^{-30} 。在 ARM9E 上整个程序不包括返回指令,占用了 31 个周期。

```

n      RN 0      ;the input angle in revolutions at Q32, result Q30
s      RN 1      ;the output sign
r      RN 2
q      RN 3
t      RN 12

cos_32 ;int cos_32(int n)
    EOR    s, n, n, LSL#1      ;cos is -ve in quadrants 1,2
    MOVS   n, n, LSL#1        ;angle in revolutions at Q33
    RSBMI  n, n, #0           ;in range 0~1/4 of a revolution
    CMP    n, #1 << 30        ;if angle < 1/8 of a revolution
    BCC    cos_core           ;take cosine
    SUBEQ  n, n, #1           ;otherwise take sine of
    RSBHI  n, n, #1 << 31     ;(1/4 revolution) - (angle)
sin_core
    ;take sine of Q33 angle n
    MOV    q, n, LSR#25       ;approximation a = (q + 0.5)/32
```

ARM 嵌入式系统开发

```

SUB    n, n, q, LSL#25      ;n = n - (q/32) at Q33
SUB    n, n, #1 << 24      ;n = n - (1/64) at Q33
LDR    t, = 0x6487ed51      ;round(2 * PI * 2^28)
MOV    r, n, LSL#3          ;r = n at Q36
SMULL  t, n, r, t           ;n = (x - a) * PI/2^31 at Q32
ADR    t, cossin_tab
LDR    q, [t, q, LSL#3]!    ;c(a) at Q30
LDR    t, [t, #4]           ;s(a) at Q30
EOR    q, q, s, ASR#31      ;correct c(a) sign
EOR    s, t, s, ASR#31      ;correct s(a) sign
SMULL  t, r, n, n           ;n^2 at Q32
SMULL  t, q, n, q           ;n * c(a) at Q30
SMULL  t, n, r, s           ;n^2 * s(a) at Q30
LDR    t, = 0xd5555556      ;round(- 2^32/6)
SUB    n, s, n, ASR#1       ;n = s(a) * (1 - n^2/2) at Q30
SMULL  t, s, r, t           ;s = - n^2/6 at Q32
ADD    n, n, q              ;n += c(a) * n at Q30
MOV    t, #0
SMLAL  t, n, q, s           ;n += - c(a) * n^3/6 at Q30
MOV    pc, lr               ;return n

sin_32                                ;int sin_32(int n)
AND    s, n, #1 << 31        ;sin is -ve in quadrants 2,3
MOVS   n, n, LSL#1           ;angle in revolutions at Q33
RSBMI  n, n, #0              ;in range 0~1/4 of a revolution
CMP    n, #1 << 30           ;if angle < 1/8 revolution
BCC    sin_core              ;take sine
SUBEQ  n, n, #1              ;otherwise take cosine of
RSBHI  n, n, #1 << 31        ;(1/4 revolution) - (angle)
cos_core                                ;take cosine of Q33 angle n
MOV    q, n, LSR#25          ;approximation a = (q + 0.5)/32
SUB    n, n, q, LSL#25      ;n = n - (q/32) at Q33
SUB    n, n, #1 << 24      ;n = n - (1/64) at Q33
LDR    t, = 0x6487ed51      ;round(2 * PI * 2^28)
MOV    r, n, LSL#3          ;r = n at Q26
SMULL  t, n, r, t           ;n = (x - a) * PI/2^31 at Q32
ADR    t, cossin_tab
LDR    q, [t, q, LSL#3]!    ;c(a) at Q30
LDR    t, [t, #4]           ;s(a) at Q30

```

```

EOR    q, q, s, ASR#31      ;correct c(a) sign
EOR    s, t, s, ASR#31      ;correct s(a) sign
SMULL  t, r, n, n            ;n^2 at Q32
SMULL  t, s, n, s            ;n * s(a) at Q30
SMULL  t, n, r, q            ;n^2 * c(a) at Q30
LDR    t, = 0x2aaaaaab      ;round(+ 2^23/6)
SUB    n, q, n, ASR#1        ;n = c(a) * (1 - n^2/2) at Q30
SMULL  t, q, r, t            ;n^2/6 at Q32
SUB    n, n, s               ;n += -sin * n at Q30
MOV    t, #0
SMLAL  t, n, s, q            ;n += sin * n^3/6 at Q30
MOV    pc, lr                ;return n

```

.cossin_tab

```

;table[2 * i] = round(2^30 * cos(a)) where a = (PI/4) * (i + 0.5)/32
;table[2 * i + 1] = round(2^30 * sin(a)) and 0 <= i < 32
DCD 0x3ffec42d, 0x00c90e90, 0x3ff4e5e0, 0x025b0caf
DCD 0x3fe12acb, 0x03ecadcf, 0x3fc395f9, 0x057db403
DCD 0x3f9c2bfb, 0x070de172, 0x3f6af2e3, 0x089cf867
DCD 0x3f2ff24a, 0x0a2abb59, 0x3eeb3347, 0x0bb6ecef
DCD 0x3e9cc076, 0x0d415013, 0x3e44a5ef, 0x0ec9a7f3
DCD 0x3de2f148, 0x104fb80e, 0x3d77b192, 0x11d3443f
DCD 0x3d02f757, 0x135410c3, 0x3c84d496, 0x14d1e242
DCD 0x3bfd5cc4, 0x164c7ddd, 0x3b6ca4c4, 0x17c3a931
DCD 0x3ad2c2e8, 0x19372a64, 0x3a2fcee8, 0x1aa6c82b
DCD 0x3983e1e8, 0x1c1249d8, 0x38cf1669, 0x1d79775c
DCD 0x3811884d, 0x1edc1953, 0x374b54ce, 0x2039f90f
DCD 0x367c9a7e, 0x2192e09b, 0x35a5793c, 0x22e69ac8
DCD 0x34c61236, 0x2434f332, 0x33de87de, 0x257db64c
DCD 0x32eefdea, 0x26c0b162, 0x31f79948, 0x27fdb2a7
DCD 0x30f8801f, 0x29348937, 0x2ff1d9c7, 0x2a650525
DCD 0x2ee3cebe, 0x2b8ef77d, 0x2dce88aa, 0x2cb2324c

```

239

7.6 字节顺序反转和位操作

本节介绍寄存器中位操作的优化方法。7.6.1 小节介绍字节顺序反转操作,这对读取存放在小端存储系统中的大端文件数据是很有用的;7.6.2 小节介绍对一个字的位变换,比

如,位取反,以及如何支持较复杂的多种位变换。6.7 节讨论了关于位流的打包和解包。

7.6.1 字节顺序反转

为了以最高效率来使用 ARM 核 32 位数据总线,就要一次装载和存储 8 位和 16 位数组的 4 字节。但是,如果一次装载多个字节,那么处理器的字节排列方式(大、小端)将会影响到它们在寄存器中的次序。如果这个次序与程序所希望的不一致,那么就需要对字节顺序进行反转。

下面的代码完成了一个字中字节排列顺序的反转。第 1 段代码使用了 2 个临时寄存器,每个字的反转在常量设置后,只占用了 3 个周期。第 2 段代码只使用了 1 个临时寄存器,对于转换单个字是很有用的。

```

n      RN 0      ;input, output words
t      RN 1      ;scratch 1
m      RN 2      ;scratch 2

byte_reverse
        MVN      n, #0x0000FF00      ;n=[ a ,b ,c ,d ]
        EOR      t, n, n, ROR #16     ;m=[ 0xFF,0xFF,0x00,0xFF]
        AND      t, m, t, LSR #8      ;t=[ a^c, b^d, a^c, b^d]
        EOR      n, t, n, ROR #8      ;t=[ 0,a^c,0,a^c]
        MOV      pc, lr               ;n=[ d ,c ,b ,a ]

byte_reverse_2reg
        EOR      t, n, n, ROR #16     ;n=[ a ,b ,c ,d ]
        MOV      t, t, LSR #8         ;t=[ a^c, b^d, a^c, b^d]
        BIC      t, t, #0xFF00       ;t=[ 0,a^c,b^d,a^c]
        EOR      n, t, n, ROR #8      ;t=[ 0,a^c,0,a^c]
        MOV      pc, lr               ;n=[ d ,c ,b ,a ]

```

在 ARM 中有桶形移位器,因此对一个字中的半字反转不需要任何额外的开销,因为它与循环右移 16 位是一样的。

7.6.2 位变换

7.6.1 小节中的字节反转是位变换(bit permutations)的一个特例。还有一些经常碰到

的其它比较重要的位变换(见表 7.3)。^{*}

表 7.3 常见的位变换

变换操作名称	变换动作
字节反转(byte reversal)	$[b_4, b_3, b_2, b_1, b_0] \rightarrow [1-b_4, 1-b_3, b_2, b_1, b_0]$
位反转(bit reversal)	$[b_4, b_3, b_2, b_1, b_0] \rightarrow [1-b_4, 1-b_3, 1-b_2, 1-b_1, 1-b_0]$
位扩散(bit spread)	$[b_4, b_3, b_2, b_1, b_0] \rightarrow [b_3, b_2, b_1, b_0, b_4]$
DES 变换(permutation)	$[b_5, b_4, b_3, b_2, b_1, b_0] \rightarrow [1-b_0, b_2, b_1, 1-b_5, 1-b_4, 1-b_3]$

- **位反转** 把位 k 和位 $(31-k)$ 交换;
- **位扩散** 隔开位使位 k 移到位 $2k$ (当 $k < 16$ 时), 或位 $(2k-31)$ (当 $k \geq 16$ 时);
- **DES 变换** DES 是一种数据加密标准, 是对大批量数据进行加密的一种常见算法。该算法在加密前后需要对数据进行 64 位的变换。

如果在手头有一些支持基本位变换操作的工具箱, 那么编写优化代码来实现这些变换就变得非常简单。本小节将介绍这些基本位变换(见表 7.4)。这些操作要比用循环来依次检查每一位快得多, 因为它们是一次处理 32 位的。

表 7.4 基本变换操作

基本操作名称	变换动作
A(互补位索引)	$[\dots, b_k, \dots] \rightarrow [\dots, 1-b_k, \dots]$
B(交换位索引)	$[\dots, b_i, \dots, b_k, \dots] \rightarrow [\dots, b_k, \dots, b_i, \dots]$
C(互补+交换位索引)	$[\dots, b_i, \dots, b_k, \dots] \rightarrow [\dots, 1-b_k, \dots, 1-b_i, \dots]$

假定要处理一个 2^k 位的值 n , 需要对 n 的各个位进行位序列改变, 那么可以用 k 位索引来表示 n 中每个位的位置: $b_{k-1}2^{k-1} + \dots + b_12 + b_0$ 。因此, 要对 32 位值进行位交换, 可以取 $k=5$ 。把在位置 $b_{k-1}2^{k-1} + \dots + b_12 + b_0$ 的位移到位置 $c_{k-1}2^{k-1} + \dots + c_12 + c_0$, 这里 c_i 表示 b_j 或 $1-b_j$ 。这个转换可表示为:

$$[b_{k-1}, \dots, b_1, b_0] \rightarrow [c_{k-1}, \dots, c_1, c_0] \quad (7.55)$$

举例来说, 表 7.3 显示了序列改变的符号和执行的动作。

它的要点是什么呢? 任何一种位变换都可以通过表格 7.4 中的 3 个基本的位变换操作组合而成。实际上, 只需要前 2 个操作, 因为 C 是 B 和 A 组合操作的结果。然而, 直接执行 C 可以更快速得到结果。

* 表 7.3 中的 5 位二进制数 $(b_4, b_3, b_2, b_1, b_0)$ 表示 32 位数的特定位序号。——译者注

7.6.2.1 位变换宏

下面的宏实现了一个 32 位字 n 的 3 个基本位变换操作。如果常量值已经存放在寄存器中,那么每个变换只需要 4 个周期。更大或者更小宽度的变换,可以用同样的思想。

```

mask0    EQU    0x55555555    ;set bit positions with b0 = 0
mask1    EQU    0x33333333    ;set bit positions with b1 = 0
mask2    EQU    0x0F0F0F0F    ;set bit positions with b0 = 0
mask3    EQU    0x00FF00FF    ;set bit positions with b3 = 0
mask4    EQU    0x0000FFFF    ;set bit positions with b4 = 0

MACRO
PERMUTE_A $ k
;[...b_k...] -> [...1-b_k...]
IF $ k = 4
    MOV    n,n,ROR#16
ELSE
    LDR    m, = mask $ k
    AND    t,m,n,LSR#(1 << $ k)    ;get bits with index b_k = 1
    AND    n,n,m                    ;get bits with index b_k = 0
    ORR    n,t,n,LSL#(1 << $ k)    ;swap them over
ENDIF
MEND

MACRO
PERMUTE_B $ j, $ k
;[...b_j...] -> [...b_k..b_j...] and j>k
LDR    m, = (mask $ j,AND,;NOT;mask $ k)    ;set when b_j = 0 b_k = 1
EOR    t,n,n,LSR#(1 << $ j) - (1 << $ k)
AND    t,t,m                    ;get bits where b_j!=b_k
EOR    n,n,t,LSL#(1 << $ j) - (1 << $ k)    ;change if b_j = 1 b_k = 0
EOR    n,n,t                    ;change when b_j = 0 b_k = 1
MEND

MACRO
PERMUTE_C $ j, $ k
;[...b_j..b_k...] -> [...1-b_k..1-b_j...] and j>k
LDR    m, = (mask $ j,AND;mask $ k)    ;set when b_j = 0 b_k = 0
EOR    t,n,n,LSR#(1 << $ j) + (1 << $ k)
AND    t,t,m                    ;get bits where b_j == b_k

```



```

EOR      n,n,t,LSL#(1 << $j) + (1 << $k)    ;change if bj = 1 bk = 1
EOR      n,n,t                                ;change when b_j = 0 b_k = 0
MEND

```

7.6.2.2 位变换例子

下面是这些宏在实际中的应用。位反转操作把位 b 移到位 $(31-b)$ 。换句话说,就是把索引 b 的 5 位二进制表示中的每一位进行取反,所得到的新的索引表示位 $(31-b)$ 。用 5 个基本位变换 A 可以实现位反转,就是依次把表示索引的每个位在逻辑上取反。

```

bit_reverse      ;n = [ b4   b3   b2   b1   b0 ]
PERMUTE_A 0      ;-> [ b4   b3   b2   b1   1-b0 ]
PERMUTE_A 1      ;-> [ b4   b3   b2   1-b1 1-b0 ]
PERMUTE_A 2      ;-> [ b4   b3   1-b2 1-b1 1-b0 ]
PERMUTE_A 3      ;-> [ b4   1-b3 1-b2 1-b1 1-b0 ]
PERMUTE_A 4      ;-> [ 1-b4 1-b3 1-b2 1-b1 1-b0 ]
MOV            pc, lr

```

使用变换 B 可实现难度更大的位扩散变换。这个过程忽略常数设置,只占用了 16 个周期——比使用一次测试一位的循环快得多。

```

bit_spread      ;n = [ b4 b3 b2 b1 b0 ]
PERMUTE_B 4,3   ;-> [ b3 b4 b2 b1 b0 ]
PERMUTE_B 3,2   ;-> [ b3 b2 b4 b1 b0 ]
PERMUTE_B 2,1   ;-> [ b3 b2 b1 b4 b0 ]
PERMUTE_B 1,0   ;-> [ b3 b2 b1 b0 b4 ]
MOV            pc, lr

```

最后,使用变换 C,可以在相同的周期里,同时进行位反转和位扩散。

```

bit_rev_spread  ;n = [ b4   b3   b2   b1   b0 ]
PERMUTE_C 4,3   ;-> [ 1-b3 1-b4   b2   b1   b0 ]
PERMUTE_C 3,2   ;-> [ 1-b3 1-b2   b4   b1   b0 ]
PERMUTE_C 2,1   ;-> [ 1-b3 1-b2 1-b1 1-b4   b0 ]
PERMUTE_C 1,0   ;-> [ 1-b3 1-b2 1-b1 1-b0   b4 ]
MOV            pc, lr

```

7.6.3 ‘1’位计数

‘1’位计数(bit population count)就是计算一个字中位值是 1 的个数。比如,在一个中断屏蔽中它可以很方便地找到中断设置位的数目。如果累加和不相互干涉,那么 ADD 指

ARM 嵌入式系统开发

令可以被用来并行地对各个位求和。相比较而言,用一个循环来检测各个位,速度会很慢。3 位分组计算(Divide by three and conquer)方法的思想是,把 32 位字划分成 3 位的组,每一组的和是 2 位大小的数,范围在 0~3。并行计算这些 3 位组,然后用对数方式加起来。

下面的代码实现了一个字的‘1’位计数。这个操作一共占用了 10 个周期,另加 2 个周期建立常数。

```

bit_count          ;input n = xyzxyzxyzxyzxyzxyzxyzxyzxyzxyz
LDR    m, = 0x49249249    ;01001001001001001001001001001
AND    t, n, m, LSL #1    ;x00x00x00x00x00x00x00x00x00x0
SUB    n, n, t, LSR #1    ;uuuuuuuuuuuuuuuuuuuuuuuuuuuuuu
AND    t, n, m, LSR #1    ;00z00z00z00z00z00z00z00z00z00
ADD    n, n, t            ;vv0vv0vv0vv0vv0vv0vv0vv0vv0vv
; triplets summed, uu = x + y, vv = x + y + z
LDR    m, = 0xC71C71C7    ;11000111000111000111000111000111
ADD    n, n, n, LSR #3    ;ww0vvwww0vvwww0vvwww0vvwww
AND    n, n, m            ;ww000www000www000www000www
; each www is the sum of six adjacent bits
ADD    n, n, n, LSR #6    ;sum the ws
ADD    n, n, n, LSR #12
ADD    n, n, n, LSR #24
AND    n, n, #63          ;mask out irrelevant bits
MOV    pc, lr

```

244

7.7 饱和及舍入运算

饱和运算可以把一个结果限制在一个固定的范围,以防止溢出。最常见的是 16 位或 32 位饱和运算,用下面的操作定义:

- $\text{satruatel6}(x) = x$ 限制在范围 $-0x00008000 \sim +0x00007fff$
- $\text{satruate32}(x) = x$ 限制在范围 $-0x80000000 \sim +0x7fffffff$

虽然可以很容易地把 16 位的饱和运算例子转换成 8 位或其它长度,但这些操作还是应该引起足够的关注。下面给出了标准的基本饱和及舍入(四舍五入)操作的实现。这里将使用标准的方法:对一个 32 位有符号整数 x

$$\begin{aligned}
 x \gg 31 &= \text{sign}(x) = -1 \quad (x < 0) \\
 &= 0 \quad (x \geq 0)
 \end{aligned}$$

7.7.1 饱和 32 位数到 16 位

这个操作经常在 DSP 应用中出现。比如,声音采样值在保存到存储器之前,要被饱和到 16 位。这个操作占用 3 个周期,常数 m 要预先存放在一个寄存器中。

```
;b = saturate16(b)
LDR    m, = 0x00007FFF      ;m = 0x7FFF maximum + ve
MOV     a, b, ASR#15        ;a = (b >> 15)
TEQ     a, b, ASR#31        ;if (a != sign(b))
EORNE   b, m, b, ASR#31     ;b = 0x7FFF ~ sign(b)
```

7.7.2 饱和左移

在信号处理时,左移可能会溢出,这就需要饱和这个结果。下面的操作对常量移位花费 3 个周期,对可变移位 c 花费 5 个周期。

```
;a = saturate32(b << c)
MOV     m, # 0x7FFFFFFF     ;m = 0x7FFFFFFF max + ve
MOV     a, b, LSL c         ;a = b << c
TEQ     b, a, ASR c         ;if (b != (a >> c))
EORNE   a, m, b, ASR#31     ;a = 0x7FFFFFFF ~ sign(b)
```

7.7.3 舍入右移

舍入右移对于常数移位需要 2 个周期,对于非零变量移位需要 3 个周期。

注意:一个零变量的移位只能在进位清除后才能正确执行。

```
;a = round(b >> c)
ADDS    b, b, #0            ;clear carry so works for c = 0
MOVS    a, b, ASR c         ;a = b >> c, carry = b bit c-1
ADC     a, a, #0            ;if (carry) a++ to round
```

7.7.4 饱和的 32 位加减法

在 ARMv5TE 核上,新的指令 QADD 和 QSUB 提供了饱和加法和减法。如果在 ARMv4T 或更早版本的核上,则要使用下面的代码来实现。代码需要 2 个周期,及一个寄

寄存器保存常量。

```

;a = saturate32(b + c)
MOV      m, #0x80000000      ;m = 0x80000000 max - ve
ADDS     a, b, c              ;a = b + c, V records overflow
EORVS    a, m, a, ASR # 31    ;if (V) a = 0x80000000~sign(a)

;a = saturate32(b - c)
MOV      m, #0x80000000      ;m = 0x80000000 max - ve
SUBS     a, b, c              ;a = b - c, V records overflow
EORVS    a, m, a, ASR # 31    ;if (V) a = 0x80000000~sign(a)

```

7.7.5 饱和绝对值

如果输入值是一0x80000000,那么绝对值运算就会溢出。下面 2 个周期的代码处理了这种情况:

```

;a = saturate32(abs(b))
SUB      a, b, b, LSR # 31    ;a = b - (b < 0)
EOR      a, a, a, ASR # 31    ;a = a ^ sign(a)

```

类似的原理,一个累加非饱和的绝对值也占用了 2 个周期:

```

;a = b + abs(c)
EORS     a, c, c, ASR # 32    ;a = c^sign(c) = abs(c) - (c < 0)
ADC      a, b, a              ;a = b + a + (c < 0)

```

7.8 随机数产生

要产生真正的随机数,就需要特殊的硬件作为随机噪声源;然而,对许多计算机应用来说,比如游戏和建模,产生的速度要比统计纯度重要得多。这些应用通常使用伪随机数。

伪随机数其实不是真正的随机数,只是一串重复序列生成的数。但是,这个序列足够长,也足够分散,这样产生的数似乎是随机的。典型地,通过迭代一个简单的 R_{k-1} 的函数,取伪随机数序列的第 k 个元素 R_k :

$$R_k = f(R_{k-1}) \quad (7.56)$$

为了快速生成伪随机数,需要找到一个函数 $f(x)$ 。这个函数要容易计算,并能生成看上去很随机的输出。这个序列在重复之前,也必须非常长。对一个 32 位数的序列,最大长

度显然为 2^{32} 。

一个线性一致产生器使用下面形式的函数：

$$f(x) = (a * x + c) \% m;$$

这些函数在 Knuth 所著的 *Seminumerical Algorithms* 一书 3.2.1 小节和 3.6 节中已经详细研究过了。要取得更快的计算速度,应取 $m=2^{32}$ 。Knuth 书中的理论保证了,如果 $a \% 8 = 5$ 且 $c=a$,那么产生的序列的最大长度是 2^{32} ,而且看起来似乎是随机的。比如,假定 $a=0x91e6d6a5$,那么下面的迭代就可以产生伪随机数序列:

```
MLA    r, a, r, a    ; r_k = ( a * r_(k-1) + a ) mod 2^32
```

由于 m 是 2 的乘幂,因此序列的低阶位并不是很随机的。应使用高阶位来得到更小范围的伪随机数。比如,设 $s=r \gg 28$,产生一个在范围 0~15 的 4 位的随机数 s 。更一般地,下面的代码产生一个 0~ n 的伪随机数:

```
; r is the current random seed
; a is the multiplier (eg 0x91E6D6A5)
; n is the random number range (0...n-1)
; t is a scratch register
MLA    r, a, r, a    ; iterate random number generator
UMULL  t, s, r, n    ; s = (r * n) / 2^32
; r is the new random seed
; s is the random result in range 0...n-1
```

7.9 总 结

ARM 指令只能实现简单的基本运算,比如加法、减法和乘法。要执行更多复杂的操作比如除法、平方根和三角函数,就要编写软件程序来完成。有许多有用的技巧和算法,可以改善这些复杂操作的性能。本章介绍的算法和例子,都是针对一些标准操作的。

这些技巧包括:

- 使用二分搜索或试探减法来计算小的商;
- 使用 Newton-Raphson 迭代快速计算开方和倒数;
- 使用查表和级数展开来计算超越函数,如 \exp , \log , \sin 和 \cos ;
- 使用带桶形移位的逻辑操作来执行位交换,而不是单独检测各个位;
- 使用乘累加指令来生成伪随机数。

第 8 章

数字信号处理

- 表示一个数字信号
- 基于 ARM 的 DSP 介绍
- FIR 滤波器
- IIR 滤波
- 离散傅里叶变换
- 总 结

8 数字信号处理

现在微处理器已经有足够的计算能力来处理实时数字信号,如大家熟悉的 MP3 音频播放器、数码相机、数字移动或蜂窝电话等。数字信号处理需要足够的存储器带宽和快速的乘累加运算。本章将介绍一些方法,使 ARM 处理器在数字信号处理(DSP)应用上能获得最好的性能。

一个传统的嵌入式或便携式设备一般包括 2 类处理器:一个微控制器处理用户接口;另一个独立的 DSP 处理器处理数字信号,如音频。然而,由于出现了更好性能和更高时钟频率的处理器,现在开发者能够使用单个处理器来处理上述的 2 项任务。相对于双核的方案,单核设计能够减少费用和降低功耗。

ARM 体系结构的加强使 ARM 可以很好地适于许多 DSP 应用。在 ARM9E 及以后的 ARM 核中使用的 ARMv5TE,提供了高效的乘累加操作。通过仔细的编码,ARM9E 处理器在应用系统的数字信号处理部分可以获得高效的性能,同时在应用系统的控制部分又可以胜过 DSP。

DSP 应用一般对乘法和数据存取要求很高。一个基本的操作是乘累加,即把 2 个 16 位有符号数相乘,并把结果累加得到一个 32 位的有符号和。表 8.1 显示了不同 ARM 核性能的递增。第 2 列给出 2 个有符号 16 位数相乘、32 位累加的时钟数;第 3 列是 2 个有符号 32 位数相乘,并进行 64 位累加的时钟数。后者对于高质量音频算法如 MP3 是特别有用的。

表 8.1 假设开发者为某些任务使用最有效的指令并且避免任何乘后互锁(*postmultiply interlocks*)。8.2 节将对此作详细描述。

表 8.1 处理器的乘累加操作时间

处理器(体系结构)	带 32 位累加的 16×16 位乘法(周期数)	带 64 位累加的 32×32 位乘法(周期数)
ARM7(ARMv3)	~12	~44
ARM7TDMI(ARMv4T)	4	7
ARM9TDMI(ARMv4T)	4	7
StrongARM(ARMv4T)	2 or 3	4 or 5
ARM9E(ARMv5TE)	1	3
Xscale(ARMv5TE)	1	2~4
ARM1136(ARMv6)	0.5	2 (结果在高半部分)

由于 DSP 算法对数据带宽和性能的要求较高,用户经常要用汇编语言来实现这些算法,并且需要对寄存器分配和指令调度进行精细控制,从而获得最好的性能。本章无法覆盖所有 DSP 算法的实现,所以把注意力集中在一些能用到整个 DSP 算法的通用例子和一般规则。

8.1 节将讨论在 ARM 中信号表示的基本问题,以便更好地处理它;8.2 节介绍在 ARM

ARM 嵌入式系统开发

上实现 DSP 算法的一般规则。

滤波可能是最常用的信号处理操作,它可以被用来进行噪声消除、信号分析或者信号压缩。8.3 节和 8.4 节将详细讨论音频滤波。另外一个常用的算法是离散傅里叶变换(DFT)。这是一个把信号从时域转成频域,或从频域转成时域的变换。8.5 节将讨论 DFT。

8.1 表示一个数字信号

在处理数字信号之前,首先需要选择一种信号的表示方法。如何在只有整数类型的 ARM 处理器上描述一个信号呢?这是一个会影响 DSP 软件设计的重要问题。本章中使用符号 x_t 和 $x[t]$ 表示信号 x 在时刻 t 的值。第 1 个符号一般在等式和公式中更清楚些,第 2 个符号用在编程例子中,它看起来更加接近 C 语言中的数组符号。

8.1.1 选择一种表示方法

在模拟信号 $x[t]$ 中,标号 t 和值 x 都是连续的实变量。要把一个模拟信号转成数字信号,必须选择有限个采样点 t_i 和采样值 $x[t_i]$ 的数字表示方法。

图 8.1 表示一个正弦波信号在采样点 0,1,2,3 等处被数字化。这样的信号在音频处理中是很典型的,其中 $x[t]$ 表示第 t 个音频采样值。

例如:在 CD 播放器中,采样频率是 44 100 Hz(即采样 44 100 次每秒)。因此 t 表示以采样周期 $1/44\,100\text{ Hz} = 22.7\text{ }\mu\text{s}$ 为单位的时间。在这个应用中, $x[t]$ 表示扬声器在时间 t 时对应的正负电压。

当选择 $x[t]$ 的表示时,有 2 个问题须注意:

- **信号的动态范围**——式(8.1)定义的信号的最大波动。对一个有符号信号,是其可能的最大绝对值。对这个例子,令 $M=1\text{ V}$ 。

$$M = \max(|x[t]|) \text{ 对所有 } t = 0, 1, 2, 3 \dots \quad (8.1)$$

- **所选择的表示法的精度**,有时以最大值范围的比值方式给出。例如,百万分之 100(1×10^{-4})的精度意味着每个 $x[t]$ 必须在式(8.2)的误差范围内被表示:

$$E = M \times 0.0001 = 0.0001\text{ V} \quad (8.2)$$

需要找到最好的方法来存储 $x[t]$,以满足给定的动态范围和精度要求。

可以用一个浮点数表示 $x[t]$,这样肯定可以满足动态范围和精度的要求,并且也会很

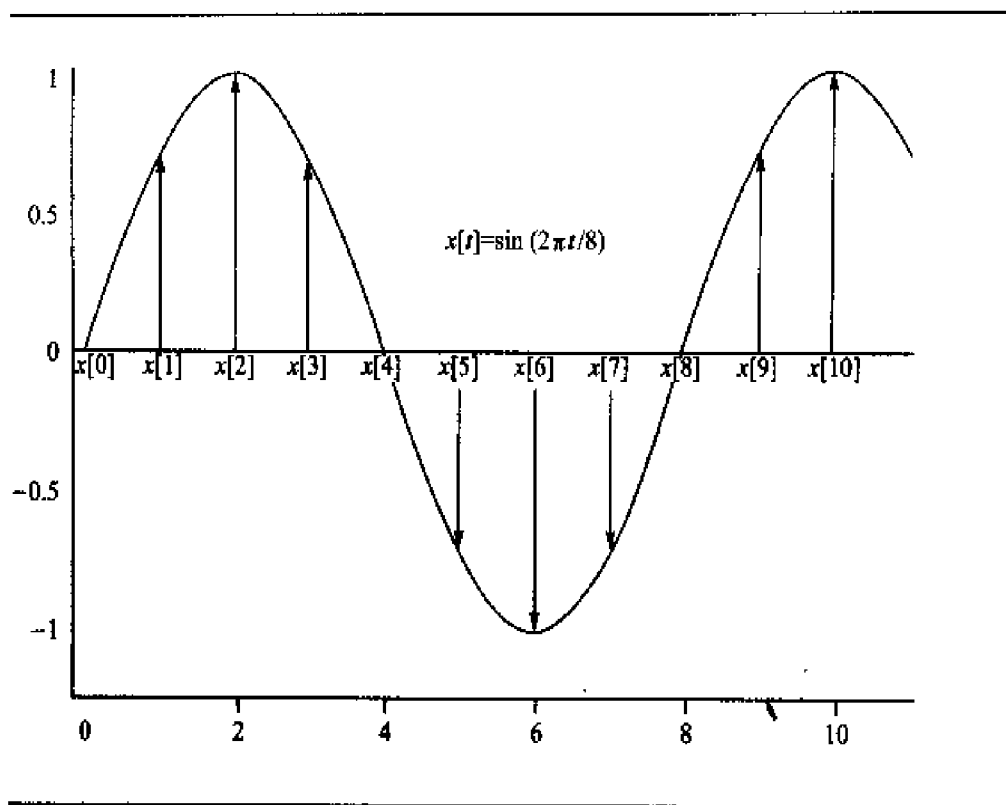


图 8.1 数字化一个模拟信号

容易使用 C 的浮点类型来操作;然而,大多数 ARM 核硬件不支持浮点,所以这样的浮点表示会非常慢。

一种快速编码的较好选择是定点表示。定点表示使用一个被分数等比的整数来表示一个分数值。例如:对一个 0.000 1 V 的最大误差,在每个表示值间只需间隔 0.000 2 V。这就可以用式(8.3)定义的整数 $X[t]$ 来表示 $x[t]$:

$$X[t] = \text{最接近的整数}(5\,000 \times x[t]) \quad (8.3)$$

实际上可以用一个 2 的幂来作等分,通过移位作等分来实现乘法和除法。在这种情况下,比 5 000 大的 2 的幂的最小值是 $2^{13} = 8\,192$ 。如式(8.4)所示,称 $X[t]$ 是 $x[t]$ 的 Qk 定点表示:

$$X[t] = \text{最接近的整数}(2^k x[t]) \quad (8.4)$$

在这个例子中,可使用 Q13 表示方法,以满足精度要求,因为 $x[t]$ 的范围为 $-1 \sim +1$ V, $X[t]$ 的范围为 $-8\,192 \sim 8\,192$ 。C 语言中 16 位 short 类型变量的值域可以覆盖这个范围。幅值在 $-1 \sim +1$ V 之间变化的信号经常以 Q15 存储,因为这样一个 short 类型的整数可以等分信号的最大范围是 $-32\,768 \sim +32\,767$ 。需要注意的是, $+1$ V (最大绝对值 M) 没有一个确切的表示方式,可以用 $+32\,767$ 表示的 $(1 - 2^{-15})$ 近似地表示 $+1$ V (M)。然而,在 8.1.2 小节中将会发现,扩大到最大范围表示并非都是好主意,这样会增加在定点表示法操

作时溢出的可能性。

在定点表示中,用整数表示每个信号值,并为整个信号使用同样的比例刻度。这与一个浮点表示法不同——每个信号值 $x[t]$ 的浮点表示都有各自依赖于 t 的比例刻度,即指数。

一个常见的错误是,认为浮点比定点精确。对于同样的比特数,定点表示更精确;而浮点表示则以较低的绝对精度为代价,提供了更大的动态范围。例如:如果使用 32 位整数保持一个定点值来比例划分全部范围(满量程),那么一个表示值的最大误差是 2^{-32} 。然而,单精度 32 位浮点值将带来 2^{-24} 的相对误差。单精度浮点的尾数是 24 位,尾数开始的 1 不被存储,所以实际上只用了存储的 23 位。对于在最大数附近的值,定点表示精确了 $2^{32-24} = 256$ 倍。当最大误差比相对精度更重要时,8 位的浮点指数很少被使用。

总之,当信号范围有明确的边界,而最大误差又是重要因素时,定点表示是最好的。当信号没有明确的边界,并且需要较大的动态范围时,浮点表示比较好。还可以使用后面介绍的其它表示法,它们比定点表示能提供更大的动态范围,也比浮点表示实现起来更有效。

8.1.1.1 饱和定点表示

如果信号的最大值是不知道的,但大部分采样值都分布在一个确定的范围内,在这种情况下,则可以使用一种基于一般范围的定点表示,把任何超出范围的采样值,饱和或消减到正常范围内最接近的采样值。这种方法以某些非常大的信号失真为代价,获得较高的精度。有关饱和运算的内容可以参考 7.7 节。

8.1.1.2 块浮点表示

当小采样值接近大采样值时,它们通常是不重要的。在这种情况下,可以把信号分成采样块或帧,依照信号在每块或帧中的不同强度,使用不同的定点比例刻度。

这和浮点表示有些类似,不同的是,在块浮点表示中,每一帧内的所有采样值具有相同的指数;而在浮点表示中,每个采样值都有自己的指数。这样,就可以使用有效的定点操作来操作同一帧内的采样值,而仅仅在帧间数值比较时,需要一定代价的指数相关操作。

8.1.1.3 对数表示法

如果信号 $x[t]$ 有很大的动态范围,并假设乘法运算远比加法运算频繁,那么可以使用以 2 为底的对数表示方法。对于这种方式,考虑相关信号 $y[t]$:

$$y[t] = \text{lb}(x[t]) \quad (8.5)$$

用定点格式表示 $y[t]$ 。

$$\begin{aligned} \text{用} & & y[a] &= y[b] + y[c] \\ \text{替换} & & x[a] &= x[b] \times x[c] \end{aligned} \quad (8.6)$$

$$\begin{aligned} \text{并用} & & y[a] &= y[b] + \text{lb}(1 + 2^{y[c] - y[b]}) \\ \text{替换} & & x[a] &= x[b] \times x[c] \end{aligned} \quad (8.7)$$

在第2种情况下,令 $y[c] \leq y[b]$ 。用查表和(或)插值的方法计算函数 $f(x) = \log_2(1+2^x)$ 。参考7.5节中有关 $\log_2 x$ 和 2^x 的有效实现。

8.1.2 操作以定点格式存储的值

设现在选择 Q_k 定点表示信号 $x[t]$ 。换句话说,有一个整数 $X[t]$ 的数组,如:

$$X[t] = \text{最接近}(2^k x[t]) \text{ 的整数} \quad (8.8)$$

同样,如果用二进制符号来表示整数 $X[t]$,并在 k 位和 $k-1$ 位之间插入一个二进制小数点,那么就有 $x[t]$ 的值。例如:在图8.2中, Q_{15} 的定点值 $0x6000$ 表示二进制数 0.11 ,或者是十进制数 $3/4=0.75$ 。

位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6000=	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

图8.2 3/4的 Q_{15} 形式定点表示法

后续章节包含适用于定点信号处理的一些基本操作,如:加法、减法、乘法、除法和平方根。以下是适用于所有定点操作的几个概念:

- **右移舍入** 当使用右移来做与2的幂的除法时,移位舍入应该向 $-\infty$ 方向,而不舍入到最接近的整数。更确切的表达是,用 $y = (x + (1 \ll \text{shift} - 1)) \gg \text{shift}$ 代替。这样就向上舍去0.5,舍入到最接近的整数。有效的具体实现请参考7.3.3小节。
- **除法舍入** 对于无符号的除法,使用 $y = (x + (d \gg 1)) / d$,而不用 $y = x / d$ 。这样得到一个舍入的结果。
- **净空间(headroom)** 定点表示法的净空间是指,可以在整数中被存储的最大数值和可能出现的最大数值的比值。例如:假如使用16位整数存储一个 Q_{13} 形式的音频信号,该信号的范围为 $-1 \sim +1$,那么就有净空间4倍或2位。此时,也可以对采样点作2次倍乘而不用担心会溢出。
- **Q 形式的转换** 如果 $X[t]$ 是 $x[t]$ 的一个 Q_n 表示,那么

$$X[t] \ll k \text{ 是 } x[t] \text{ 的一个 } Q_{(n+k)} \text{ 形式} \quad (8.9)$$

$$X[t] \gg k \text{ 是 } x[t] \text{ 的一个 } Q_{(n-k)} \text{ 形式} \quad (8.10)$$

在下面的章节中,对于确定的信号 $x[t]$, $c[t]$ 和 $y[t]$,用 $X[t]$, $C[t]$, $Y[t]$ 分别表示它们的 Q_n , Q_m , Q_d 定点形式。

8.1.3 定点信号的加法和减法

通常情况是把信号

$$y[t] = x[t] + c[t] \quad (8.11)$$

转换成定点格式,即近似地:

$$Y[t] = 2^d y[t] = 2^d (x[t] + c[t]) = 2^{d-n} X[t] + 2^{d-m} C[t] \quad (8.12)$$

或者用整数 C

$$Y[t] = (X[t] \ll (d-n)) + (C[t] \ll (d-m));$$

这里使用一个约定,把一个负的左移值解释为右移舍入。换句话说,就是先转换 $x[t]$ 和 $c[t]$ 为 Qd 形式,然后加起来给 $Y[t]$ 。

在编译时,已经知道 d, n 和 m 的值,所以要决定移位方向或者是否移位是没有问题的。实际上通常令 $n=m=d$, 因此整数加法给出一个定点加:

$$Y[t] = X[t] + C[t]$$

倘若 $d=m$ 或 $d=n$, 可以使用 ARM 的桶形移位器来完成这个操作:

$$Y[t] = X[t] + (C[t] \ll (d-m)); \quad /* d == n \text{ 的情况} */$$

$$Y[t] = C[t] + (X[t] \ll (d-n)); \quad /* d == m \text{ 的情况} */$$

但必须注意,上面的等式只有在移位值和结果不溢出的情况下才有效。例如,如果 $Y[t] = X[t] + C[t]$, 即 $Y[t]$ 的动态范围是 $X[t]$ 和 $C[t]$ 动态范围的和,则很容易溢出整数的范围。

一般有 4 种防止溢出的方法:

- 确保 $X[t]$ 和 $C[t]$ 都有一位的净空间。换句话说,每个值只用了其整数的一半范围,这样在加法中就不会溢出。
- 对于 Y , 使用比 X 和 C 更大的值域范围的类型。例如,如果 $X[t]$ 和 $C[t]$ 使用 16 位整数类型存储,那么 $Y[t]$ 可以用一个 32 位整数类型,这样就可以保证不会溢出。实际上,这样 $Y[t]$ 就有 15 位的净空间,所以可以对 $Y[t]$ 加上很多 16 位值而不会发生溢出。
- 对 $y[t]$ 使用一个较小的 Q 表示法。例如:如果 $d=n-1=m-1$, 那么运算变为:

$$Y[t] = (X[t] + X[t-1]) \gg 1;$$

因为移位在加法之后,这个操作需要 2 个周期。然而,运算结果不会溢出。

- 使用饱和。如果 $X[t] + C[t]$ 的结果超出了存储 $Y[t]$ 的整数的范围,那么把这个值缩减为在这个范围内的最接近的可能值(即饱和运算)。7.7 节论述了如何有效地

实现饱和运算。

8.1.4 定点信号的乘法

通常情况是把信号表达式

$$y[t] = x[t]c[t] \quad (8.13)$$

转换成定点格式,即近似地:

$$Y[t] = 2^d y[t] = 2^d (x[t]c[t]) = 2^{d-n-m} X[t]C[t] \quad (8.14)$$

或者是整数 C :

$$Y[t] = (X[t] * C[t]) \gg (n+m-d);$$

这里把一个负的左移值解释为右移舍入。 $X[t]C[t]$ 的结果是一个 $Y[t]$ 的 $Q(n+m)$ 表示,移位后转换成 Qd 形式。有 2 种通常的应用:

- 一系列乘法的累加。在这种情况下,假设 $d=n+m$,使用一个比 $X[t]$ 和 $C[t]$ 宽的整数来存 $Y[t]$ 。乘法和乘累加运算就是:

```
Y[t] = X[t] * C[t];    /* 乘法 */
Y[t] += X[t] * C[t];  /* 乘累加 */
```

- 信号 $Y[t]$ 是信号 $X[t]$ 和一些比例系数的逐点乘法。这种情况下,令 $d=n$,那么运算是:

$$Y[t] = (X[t] * C[t]) \gg m;$$

对于音频 DSP 应用, 16×16 位乘法经常使用,一般 n 和 m 的值是 14 和 15。在做加法和减法时,要检查每个运算,确保不会溢出。

【例 8.1】 设 $X[t]$ 是音频信号 $x[t]$ 的 16 位有符号表示。假设要把信号的能量减少一半,这样必须用 $1/\sqrt{2}$ 乘以每个采样值,所以 $c[t] = 2^{-0.5} = 0.707\ 106\ 78\dots$

既然用 16 位表示 $X[t]$, 16 位乘法将足够。最大的 2 的指数是 15, 这个指数可乘以 $c[t]$ 并且使它保持 16 位整数。所以,令 $n=d, m=15, C[t] = 2^{15}/(\sqrt{2}) = 23\ 710 = 0x5A82$, 因此可以使用整数操作来比例划分

$$X[t] = (X[t] * 0x5A82) \gg 15$$

8.1.5 定点信号的除法

通常情况是把信号表达式

$$y[t] = \frac{x[t]}{c[t]} \quad (8.15)$$

转换成定点格式,即近似地:

$$Y[t] = 2^d y[t] = \frac{2^d x[t]}{c[t]} = 2^{d-n+m} \frac{X[t]}{C[t]} \quad (8.16)$$

或者是整数 C :

$$Y[t] = X[t] \ll (d - n + m) / C[t];$$

同样是一个负的左移表示为右移。

必须注意:左移不会引起溢出。

在典型的应用中, $n=m$, 那么上面的运算得到一个精度在二进制小数 d 位的 Q_d 结果:

$$Y[t] = (X[t] \ll d) / C[t];$$

请参考 7.3.3 小节关于定点除法的有效实现。

8.1.6 定点信号的平方根

通常情况是把信号表达式

$$y[t] = \sqrt{x[t]} \quad (8.17)$$

转换成定点格式,即近似地:

$$Y[t] = 2^d y[t] = 2^d \sqrt{x[t]} = \sqrt{2^{2d-n} X[t]} \quad (8.18)$$

或者是整数 C :

$$Y[t] = \text{isqrt}(X[t] \ll (2 * d - n));$$

函数 `isqrt` 得到整数平方根的最接近整数。请参考 7.4 节有关平方根的有效实现。

8.1.7 小结:数字信号的表示

要选择一个信号值的表示,可以使用以下规则:

- 在原型算法中使用浮点表达方式。对速度要求苛刻的应用,不要使用浮点表示。大

多数 ARM 实现不包含硬件浮点支持。

- 在对速度要求苛刻而动态范围要求适中的应用中,使用定点表示。ARM 核 8, 16, 32 位定点 DSP 提供了很好的支持。
- 对速度和动态范围都有较高要求的应用,使用块浮点或对数表示法。

表 8.2 总结了在定点算法中如何实现一些标准运算。这里假设有 3 种信号 $x[t]$, $c[t]$, $y[t]$, 它们分别有 Q_n, Q_m, Q_d 形式的表示 $X[t], C[t], Y[t]$ 。换句话说:

$$X[t] = 2^n x[t], C[t] = 2^m c[t], Y[t] = 2^d y[t] \quad (8.19)$$

是最接近的整数。

为了使表格更简明,这里使用符号“ \lll ”表示一个依据移位数符号,既可左移,也可右移的运算符。通常:

$x \lll s ; =$	
$x \ll s$	如果 $s \geq 0$
$x \gg (-s)$	如果 $s < 0$ 并且不需要舍入
$(x + \text{round}) \gg (-s)$	如果 $s < 0$ 需要舍入
$\text{round} ; = (1 \ll (-1 - s))$	如果 0.5 上舍
$(1 \ll (-1 - s)) - 1$	如果 0.5 下舍

表 8.2 标准定点运算小结

信号运算	等价定点整数
$y[t] = x[t]$	$Y[t] = X[t] \lll (d - n);$
$y[t] = x[t] + c[t]$	$Y[t] = (X[t] \lll (d - n)) + (C[t] \lll (d - m));$
$y[t] = x[t] - c[t]$	$Y[t] = (X[t] \lll (d - n)) - (C[t] \lll (d - m));$
$y[t] = x[t] * c[t]$	$Y[t] = (X[t] * C[t]) \lll (d - n - m);$
$y[t] = x[t] / c[t]$	$Y[t] = (X[t] \lll (d - n + m)) / C[t];$
$y[t] = \text{sqrt}(x[t])$	$Y[t] = \text{isqrt}(X[t] \lll (2 * d - n));$

注意:必须经常检查中间结果和输出值的精度和动态范围,确保没有溢出或不可接受的精度丢失。这些考虑决定了表示法和整数的大小。

这些等式是最一般的形式。实际上,对于加法和减法,通常令 $d = n = m$ 。对于乘法,通常令 $d = n + m$ 或 $d = n$ 。由于在编译时已经知道 d, n 和 m ,因此可以消除 0 移位。

8.2 基于 ARM 的 DSP 介绍

本节首先要考虑 ARM 体系结构的特点,这对于编写 DSP 应用非常重要。然后将依次考虑每个常用 ARM 内核的实现,重点突出内核对数字信号处理的优点和弱点。

ARM 核不是专门用于 DSP 的,它没有单一的指令来实现乘累加和并行数据访问。但是,通过重复使用装载的数据,可以取得很好的 DSP 性能。关键的思想是使用块算法。这个算法一次可计算多个结果,并且相对于计算单个结果,这只需较小的存储器带宽,并可改善性能和降低功耗。

对于精度和饱和,ARM 也不同于标准的 DSP。通常,ARM 不提供自动饱和运算,各种运算的饱和通常需要额外的时钟周期。7.7 节讨论了 ARM 上的饱和操作。另一方面,ARM 很好地支持扩展精度的 32 位乘以 32 位得到 64 位结果的操作。这种操作对 CD 音质的音频应用特别重要,这些应用需要超过 16 位的中间精度(intermediate precision)。

从 ARM9 开始,ARM 实现使用了针对装载和乘法的多级执行流水线,这项技术带来了潜在的处理器互锁。如果装载一个数据,接着在后面两条指令中的任意一条使用这个数据,那么处理器可能停下来几个周期,以等待被装载的数据。同样,如果在乘法后面的指令中使用乘法的结果,那么也可能引入等待周期。优化代码,以避免这些等待,是非常重要的。关于指令优化请参考 6.3 节的讨论。

小结 ARM 上 DSP 代码的编程指导

- 由于饱和操作需要额外的 CPU 周期,在设计 DSP 算法时要避免饱和操作。可以使用扩展精度的运算或附加的比例尺度变换替代饱和操作。
- 设计 DSP 算法时,要最小化装载和存储次数。一旦装载了一个数据项,就要尽可能在多个运算中使用这个数据项。可以一次计算多个输出结果,或是把多个运算并置在一起。例如,可以同时执行一个点乘和信号尺度变换,而只装载数据一次。
- 使用 ARM 汇编,以避免处理器互锁。装载和乘法指令的结果在其后的指令中常常还没有就绪,必须增加等待周期,有时需要等待好几个周期,才能得到结果。详细的指令周期定时可参考附录 D。
- ARM 有 14 个通用寄存器: $r0 \sim r12$ 和 $r14$ 。设计 DSP 算法时,要使内部循环体使用 14 个或更少的寄存器。

在后续章节中,将依次考虑各个标准的 ARM 核,并在每个 ARM 核上实现一个点乘(点积)的例子。点乘是一个最简单的 DSP 运算,将强调在不同 ARM 核上实现的差异性。从两个信号 x_i 和 c_i 的 N 个采样值的点乘,得到一个相关的值 a :

$$a = \sum_{i=0}^{N-1} c_i x_i \quad (8.20)$$

点乘函数的 C 接口原型是:

```
int dot_product(sample * x, coefficient * c, unsigned int N);
```

这里

- sample 是一个表示 16 位音频采样的数据类型, 通常使用 short 类型;
- coefficient 是一个表示 16 位系数的数据类型, 通常使用 short 类型;
- $x[i]$ 和 $c[i]$ 是长度为 N 的 2 个数组(数据和系数);
- 函数返回 32 位整数的累加点乘结果 a 。

8.2.1 ARM7TDMI 的 DSP

ARM7TDMI 有一个单周期 32 位乘以 8 位的提前终止乘法阵列。16 位乘以 16 位得到 32 位的计算需要 4 个周期。对 0 等待的内存或 cache 指令中, 装载指令需要 3 个周期, 存储指令需要 2 个周期。ARM7TDMI 指令的时钟周期数可参考附录 D 的 D.2 节。

小结 ARM7TDMI 上 DSP 代码的编程指导

- 装载指令执行较慢, 装载一个值需要 3 个周期。要高效地访问存储器, 应该使用多寄存器装载和存储指令 LDM 和 STM。多寄存器装载和存储指令在第一个字之后, 对每个附加字的传送只需要 1 个周期, 这也意味着在 32 位字中存储 16 位数据值会更加有效。
- 乘法指令根据乘积 R_s 中的第 2 个操作数通常会提前终止。为了可预测的性能, 可以使用第 2 个操作数来表示固定系数或倍数。
- 乘法比乘累加快 1 个周期, 有时把 MLA 指令拆分成独立的 MUL 和 ADD 指令会很有用, 接着可以使用桶形移位和 ADD 来执行变尺度累加。
- 可以使用带移位的运算指令来更快地实现与固定系数的乘法。例如: $240x = (x \ll 8) - (x \ll 4)$ 。对于形如 $\pm 2^a \pm 2^b \pm 2^c$ 的固定系数, 带移位的 ADD 和 SUB 可以比 MLA 更快地实现乘法运算。

【例 8.2】显示 ARM7TDMI 的 16 位点乘的优化。

每个 MLA 需要 4 个周期。这里把 16 位输入采样值存放在 32 位的字中, 这样就可以使用 LDM 指令高效地装载。

x	RN 0	; 输入数组 x[]
c	RN 1	; 输入数组 c[]
N	RN 2	; 采样点的数目(5 的倍数)

```

acc      RN 3      ;累加器
x_0      RN 4      ;数组 x[] 的元素
x_1      RN 5
x_2      RN 6
x_3      RN 7
x_4      RN 8
c_0      RN 9      ;数组 c[] 的元素
c_1      RN 10
c_2      RN 11
c_3      RN 12
c_4      RN 14

;int dot_16by16_arm7m(int *x, int *c, unsigned N)
dot_16by16_arm7m
    STMFID    sp!, {r4-r11, lr}
    MOV      acc, #0
loop_7m ;累加 5 个乘积
    LDMIA     x!, {x_0, x_1, x_2, x_3, x_4}
    LDMIA     c!, {c_0, c_1, c_2, c_3, c_4}
    MLA      acc, x_0, c_0, acc
    MLA      acc, x_1, c_1, acc
    MLA      acc, x_2, c_2, acc
    MLA      acc, x_3, c_3, acc
    MLA      acc, x_4, c_4, acc
    SUBS     N, N, #5
    BGT      loop_7m
    MOV      r0, acc
    LDMFD    sp!, {r4-r11, pc}

```

这段代码假设采样点的数目 N 是 5 的倍数,因此可以使用 5 个字的多数据装载来增加数据的带宽,平均每次装载的开销是 $7/5=1.4$ 个周期。如果使用 LDR 或 LDRSH,则每次装载需要 3 个周期。内循环最多需要 $7+7+5\times 4+1+3=38$ 个周期来处理包含 5 个结果的每个块。这给出了 ARM7TDMI 的一个 DSP 性能评价:每个 tap(抽头)的 16 位点乘运算平均需要 $38/5=7.6$ 个周期。如果要计算多数据乘积,8.3 节给出的块滤波算法会有更好的性能。

8.2.2 ARM9TDMI 的 DSP

ARM9TDMI 和 ARM7TDMI 一样,也有一个单周期 32 位乘以 8 位的乘法阵列。然

而,装载和存储操作比 ARM7TDMI 快了很多,如果在 load 指令之后的 2 个时钟周期不使用装载值,那么其只用了 1 个时钟周期。ARM9TDMI 指令的时钟周期数可参考附录 D 的 D.3 节。

小结 ARM9TDMI 上 DSP 代码的编程指导

- 编写代码时,只要避免在 load 指令后的 2 个周期内使用装载值,装载指令就是非常快的。使用多数据装载并没有什么好处,应该把 16 位数据存储在 16 位 short 类型的数组中,使用 LDRSH 指令装载数据。
- 乘法指令根据点乘 R_s 中第 2 个操作数的值通常会提前终止。为了可预测的性能,可以使用第 2 个操作数来表示固定系数或倍数。
- 乘法和乘累加的速度相同,所以要尽量使用 MLA 指令,而不要拆分乘法和加法。
- 可以使用带移位的运算指令来更快地实现与固定系数的乘法。例如: $240x = (x \ll 8) - (x \ll 4)$ 。对于形如 $\pm 2^a \pm 2^b \pm 2^c$ 的固定系数,带移位的 ADD 和 SUB 可以比 MLA 更快地实现乘法运算。

【例 8.3】 显示 ARM9TDMI 的 16 位点乘的优化。

每个 MLA 最坏情况下只需要 4 个周期,这里把 16 位输入采样值存放在 16 位 short 类型整数中。既然使用 LDM 没有比使用 LDRSH 更好,这里使用 LDRSH,以减小内存数据的大小。

```

x      RN 0      ;输入数组 x[]
c      RN 1      ;输入数组 c[]
N      RN 2      ;采样点的数目(4 的倍数)
acc    RN 3      ;累加器
x_0    RN 4      ;数组 x[] 的元素
x_1    RN 5      ;
c_0    RN 9      ;数组 c[] 的元素
c_1    RN 10     ;

;int dot_16by16_arm9m(short *x, short *c, unsigned N)
dot_16by16_arm9m
    STMED        sp!, {r4 - r5, r9 - r10, lr}
    MOV          acc, #0
    LDRSH        x_0, [x], #2
    LDRSH        c_0, [c], #2
loop_9m ;累加 4 个乘积
    SUBS          N, N, #4

```

```

LDRSH      x_1, [x], #2
LDRSH      c_1, [c], #2
MLA        acc, x_0, c_0, acc
LDRSH      x_0, [x], #2
LDRSH      c_0, [c], #2
MLA        acc, x_1, c_1, acc
LDRSH      x_1, [x], #2
LDRSH      c_1, [c], #2
MLA        acc, x_0, c_0, acc
LDRGTSH    x_0, [x], #2
LDRGTSH    c_0, [c], #2
MLA        acc, x_1, c_1, acc
BGT        loop_9m
MOV        r0, acc
LDMFD      sp!, {r4 - r5, r9 - r10, pc}

```

这里假设采样的数目是 4 的倍数,因此可以展开循环 4 次来改善性能。编写代码时,在装载和使用装载值之间有 4 条指令。这里使用了 6.3.1 小节的预装载方法。

- 装载使用双缓冲,在装载 x_1 和 c_1 时,用 x_0 和 c_0 ,反之亦然。
- 在内循环开始前,装载初始值 x_0 和 c_0 ,初始化双缓冲处理。
- 通常在使用一对数据时,装载下一对数据,因此必须避免最后一对的装载;否则会超出数组的范围。可以使用一个循环计数器,在最后一次循环时,计数器减到 0。只有在 $N > 0$ 的条件下,才做下一次装载。

内循环中每个循环需要 28 个周期,即每个 tap(抽头)平均需要 $28/4=7$ 个周期。更有效的块滤波实现,可参考 8.3 节。

8.2.3 StrongARM 的 DSP

StrongARM 的内核 SA-1 有一个单周期有符号 32 位乘以 12 位的提前终止乘法阵列。如果想要在乘法指令的后一条指令中使用乘法结果或启动一个新的乘法,那么内核会有 1 个周期的等待。除了有符号字节和半字装载需要 2 个周期外,load 指令只需要 1 个周期,但装载值要延迟 1 个周期才可使用。详细的 StrongARM 指令的周期定时,可参考附录 D 的 D.4 节。

小结 StrongARM 上 DSP 代码的编程指导

- 应避免有符号字节和半字装载。调整代码,以避免在装载指令后的一个周期内使用

装载值。使用多数据装载没有好处。

- 乘法指令根据在点乘 R_s 中第 2 个操作数的值会提前终止。为了可预测的性能,可以使用第 2 个操作数来表示固定系数或倍数。
- 乘法和乘累加速度相同,所以应尽量使用 MLA 指令,而不要拆分乘法和加法。

【例 8.4】 显示一个 16 位点乘。

由于装载一个有符号 16 位数需要 2 个周期,对 StrongARM 来说,使用 32 位数更有效些。为了调整 StrongARM 代码,一种方法就是交叉使用装载和乘法。

```

x      RN 0      ;输入数组 x[]
c      RN 1      ;输入数组 c[]
N      RN 2      ;采样点的数目(4 的倍数)
acc    RN 3      ;累加器
x_0    RN 4      ;数组 x[]的元素
x_1    RN 5
c_0    RN 9      ;数组 c[]的元素
c_1    RN 10

;int dot_16by16_SAI(int *x, int *c, unsigned N)
dot_16by16_SAI
    STMFD    sp!, {r4 - r5, r9 - r10, lr}
    MOV      acc, #0
    LDR      x_0, [x], #4
    LDR      c_0, [c], #4
loop_sa ;累加 4 个乘积
    SUBS     N, N, #4
    LDR      x_1, [x], #4
    LDR      c_1, [c], #4
    MLA      acc, x_0, c_0, acc
    LDR      x_0, [x], #4
    LDR      c_0, [c], #4
    MLA      acc, x_1, c_1, acc
    LDR      x_1, [x], #4
    LDR      c_1, [c], #4
    MLA      acc, x_0, c_0, acc
    LDRGT    x_0, [x], #4
    LDRGT    c_0, [c], #4
    MLA      acc, x_1, c_1, acc

```

```

BGT      loop_sa
MOV      r0, acc
LDMFD    sp!, {r4-r5, r9-r10, pc}

```

这里假设采样数 N 是 4 的倍数,所以在每个循环中处理 4 个采样数据。对于 16 位系数的最坏情况,每次乘法需要 2 个周期。编写代码时已经优化,避免了所有的装载和乘法的互锁。内循环处理每 4 个点乘运算使用 19 个周期,所以每个 tap(抽头)平均需要 $19/4 = 4.75$ 个周期。

8.2.4 ARM9E 的 DSP

ARM9E 核有一个非常快的流水线乘法器阵列,只用 1 个发射周期就可以实现 32 位与 16 位的乘法;但运算结果在下个周期还不能使用,除非是在乘累加操作中的累加器使用该结果。装载和存储操作与 ARM9TDMI 具有同样的速度。有关 ARM9E 指令的周期定时,可参考附录 D 的 D.4 节。

为了访问快速的乘法器,须使用在 ARMv5TE 体系结构扩展中定义的乘法指令。对于 16 位乘以 16 位操作,使用 SMULxy 和 SMLAxy。关于所有的乘法指令列表,可参见附录 A。

小结 ARM9E 上 DSP 代码的编程指导

- ARMv5TE 体系结构的乘法操作,能够把 32 位字拆分为 16 位的半字并相乘。为了有最高的装载带宽,应使用字装载指令来装载打包的 16 位数据项。同 ARM9TDMI 一样,应调整代码,以避免装载互锁。
- 乘法操作没有提前终止,因此对于 32 位整数乘法,应该使用 MUL 和 MLA;对于 16 位值,使用 SMULxy 和 SMLAxy。
- 乘法和乘累加速度相同,所以应尽量使用 SMLAxy 指令,而不要拆分乘法和加法。

【例 8.5】显示 ARM9E 的点乘运算。

这里假设 ARM 被配置成小端(little-endian)存储系统。如果 ARM 被配置成大端(big-endian)存储系统,那么需要交换 B 和 T 指令后缀,可以像例 8.11 那样定义宏来自动完成这件事。这里使用命名规范: x_{10} 表示寄存器的高 16 位存放 x_1 ,低 16 位存放 x_0 。

```

x      RN 0      ;输入数组 x[]
c      RN 1      ;输入数组 c[]
N      RN 2      ;采样点的数目(8 的倍数)
acc    RN 3      ;累加器
x_10   RN 4      ;封装数组 x[]的元素

```

```

x_32    RN 5
c_10    RN 9      ;封装数组 c[] 的元素
c_32    RN 10

;int dot_16by16_arm9e(short * x, short * c, unsigned N)
dot_16by16_arm9e
    STMFD    sp!, {r4 - r5, r9 - r10, lr}
    MOV      acc, #0
    LDR      x_10, [x], #4
    LDR      c_10, [c], #4
loop_9e ;累加 8 个乘积
    SUBS     N, N, #8
    LDR      x_32, [x], #4
    SMLABB   acc, x_10, c_10, acc
    LDR      c_32, [c], #4
    SMLATT   acc, x_10, c_10, acc
    LDR      x_10, [x], #4
    SMLABB   acc, x_32, c_32, acc
    LDR      c_10, [c], #4
    SMLATT   acc, x_32, c_32, acc
    LDR      x_32, [x], #4
    SMLABB   acc, x_10, c_10, acc
    LDR      c_32, [c], #4
    SMLATT   acc, x_10, c_10, acc
    LDRGT    x_10, [x], #4
    SMLABB   acc, x_32, c_32, acc
    LDRGT    c_10, [c], #4
    SMLATT   acc, x_32, c_32, acc
    BGT      loop_9e
    MOV      r0, acc
    LDMFD    sp!, {r4 - r5, r9 - r10, pc}

```

这里假设 N 是 8 的倍数, 循环展开 8 次。每次装载指令读取 2 个 16 位值, 这样可以得到较高的存储器访问带宽。内部循环需要 20 个周期来累加 8 个乘积, 每个 tap(抽头)平均需要 $20/8=2.5$ 个周期。后述的块滤波算法会给出更有效的实现方法。

8.2.5 ARM10E 的 DSP

和 ARM9E 一样, ARM10E 也是 ARMv5TE 体系结构的实现。除了 16 位乘累加需要

2 个周期外,乘法操作的范围和速度与 ARM9E 相同。有关 ARM10E 指令的周期定时,可参考附录 D 的 D.6 节。

ARM10E 实现了一个后台装载机制来加速多数据装载和存储。一个多数据装载和存储指令可以用 1 个周期发射,操作将会在后台运行。如果试图在后台装载完成之前使用数据值,那么内核会插入等待周期。ARM10E 使用一个 64 位宽度的数据通路,每个后台周期能够传送 2 个寄存器。如果地址不是 64 位对齐,那么第 1 个周期只能传送 32 位。

小结 ARM10E 上 DSP 代码的编程指导

- 多数据装载和存储在后台执行,可获得较高的存储器带宽。无论何时,应尽可能使用多数据装载和存储。仔细调整代码,在后台装载完成之前不要使用数据。
- 确保数据数组是 64 位对齐的,使装载和存储操作能够每周期传送 2 个字。
- 乘法操作不能提前终止,因此应使用 MUL 和 MLA 来实现 32 位整数乘法;16 位乘法使用 SMULxy 和 SMLAxy。
- SMLAxy 比 SMULxy 多一个周期,所以把乘累加拆分成单独的乘法和加法是有用的。

【例 8.6】 采样数 N 是 10 的倍数。

```

x          RN 0          ;输入数组 x[]
c          RN 1          ;输入数组 c[]
N          RN 2          ;采样点的数目(10 的倍数)
acc        RN 3          ;累加器
x_10       RN 4          ;封装数组 x[]的元素
x_32       RN 5
x_54       RN 6
x_76       RN 7
x_98       RN 8
c_10       RN 9          ;封装数组 c[]的元素
c_32       RN 10
c_54       RN 11
c_76       RN 12
c_98       RN 14

;int dot_16by16_arm10(short * x, short * c, int n)
dot_16by16_arm10
    STMED    sp!, {r4-r11, lr}
    LDMIA    x!, {x_10, x_32}
    MOV      acc, #0

```



```

        LDMIA      c!, {c_10, c_32}
loop_10 ;累加 10 个乘积
        SUBS      N, N, #10
        LDMIA      x!, {x_54, x_76, x_98}
        SMLABB     acc, x_10, c_10, acc
        SMLATT     acc, x_10, c_10, acc
        LDMIA      c!, {c_54, c_76, c_98}
        SMLABB     acc, x_32, c_32, acc
        SMLATT     acc, x_32, c_32, acc
        LDMGTIA    x!, {x_10, x_32}
        SMLABB     acc, x_54, c_54, acc
        SMLATT     acc, x_54, c_54, acc
        SMLABB     acc, x_76, c_76, acc
        LDMGTIA    c!, {c_10, c_32}
        SMLATT     acc, x_76, c_76, acc
        SMLABB     acc, x_98, c_98, acc
        SMLATT     acc, x_98, c_98, acc
        BGT        loop_10
        MOV        r0, acc
        LDMFD      sp!, {r4 - r11, pc}

```

内循环处理 10 个采样值共需 25 个周期,平均为 2.5 个周期/tap。

267

8.2.6 Intel Xscale 的 DSP

Intel Xscale 和 ARM9E, ARM10E 一样,是 ARM 体系结构 ARMv5TE 版本的实现。装载和乘法指令的时间与 ARM9E 相同,而且为 ARM9E 优化的代码能在 Xscale 上有效执行。有关 Xscale 指令的周期定时,可参考附录 D 的 D.7 节。

小结 Intel Xscale 上 DSP 代码的编程指导

- 双字装载指令 LDRD 在单个周期里可传送 2 个字。可调整代码,以确保在装载后的 2 个周期内不使用第 1 个装载值,以及在装载后的 3 个周期不使用第 2 个装载的寄存器。
- 应确保数据数组是 64 位对齐的,以便能使用 64 位装载指令 LDRD。
- 乘法结果不能立即获得。在乘法后面紧跟另一个乘法可能会插入等待,可调整代码,以使乘法指令插入到装载指令中,防止处理器等待。
- 乘法操作不会提前终止,因此应使用 MUL 和 MLA 来实现 32 位整数乘法;16 位乘

法使用 SMULxy 和 SMLAxy。

【例 8.7】 使用 LDRD 指令提高装载带宽。

输入数组必须是 64 位对齐的,采样数 N 是 8 的倍数。

x RN 0 ;输入数组 x[] (64 位对齐)

c RN 1 ;输入数组 c[] (64 位对齐)

N RN 2 ;采样点的数目(8 的倍数)

acc0 RN 3 ;累加器

acc1 RN 14

x_10 RN 4 ;封装数组 x[] 的元素

x_32 RN 5

x_54 RN 6

x_76 RN 7

c_10 RN 8 ;封装数组 c[] 的元素

c_32 RN 9

c_54 RN 10

c_76 RN 11

dot_16by16_xscale

STMFD sp!, {r4 - r11, lr}

LDRD x_10, [x], #8 ;预取 x_10, x_32

LDRD c_10, [c], #8 ;预取 c_10, c_32

MOV acc0, #0

MOV acc1, #0

loop_xscale

 ;累加 8 个乘积

SUBS N, N, #8

LDRD x_54, [x], #8 ;装载 x_54, x_76

SMLABB acc0, x_10, c_10, acc0

SMLATT acc1, x_10, c_10, acc1

LDRD c_54, [c], #8 ;装载 c_54, c_76

SMLABB acc0, x_32, c_32, acc0

SMLATT acc1, x_32, c_32, acc1

LDRGTD x_10, [x], #8 ;装载 x_10, x_32

SMLABB acc0, x_54, c_54, acc0

SMLATT acc1, x_54, c_54, acc1

LDRGTD c_10, [c], #8 ;装载 c_10, c_32

```

SMLABB    acc0, x_76, c_76, acc0
SMLATT    acc1, x_76, c_76, acc1
BGT       loop_xscale
ADD       r0, acc0, acc1
LDMFD     sp!, {r4 - r11, pc}

```

内循环共需 14 个周期来累加 8 个乘积, 平均每个 tap(抽头) 占用 1.75 个周期。

8.3 FIR 滤波器

有限冲激响应(the finite impulse response, FIR)滤波器是很多 DSP 应用的基本构造模块, 值得进行仔细研究。FIR 滤波可用来去除不需要的频率成分, 增强某种特定的频率, 或实现特别的效果。下面将集中讨论在 ARM 上滤波器的有效实现。FIR 滤波是数字滤波的最简单类型。滤波后的采样 y_t 线性依赖于一个固定的、有限个的非滤波采样 x_t 。以 M 表示滤波器的长度, 那么对于一些滤波系数 c_i :

$$y_t = \sum_{i=0}^{M-1} c_i x_{t-i} \quad (8.21)$$

一些书中将系数 c_i 叫做冲激响应。如果将冲激信号 $x = (1, 0, 0, 0, \dots)$ 送入滤波器, 那么输出是滤波系数 $y = (c_0, c_1, c_2, \dots)$ 的信号。

接下来考虑输出信号的动态范围和可能溢出的问题。假设使用 Q_n 和 Q_m 定点形式 $X[t]$ 和 $C[i]$ 分别表示 x_t 和 c_i 。换句话说:

$$X[t] = \text{round}(2^n x_t), C[i] = \text{round}(2^m c_i) \quad (8.22)$$

通过计算累加值 $A[t]$ 实现滤波:

$$A[t] = C[0]X[t] + C[1]X[t-1] + \dots + C[M-1]X[t-M+1] \quad (8.23)$$

接着 $A[t]$ 是 y_t 的一个 $Q(n+m)$ 表示。但是, $A[t]$ 是多大呢? 要确保 $A[t]$ 不溢出它的整数范围, 并且不会得到一个没有意义的滤波结果, 究竟需要多少位的精度呢? 这里有 2 个非常有用的不等式来回答这个问题。

$$|A[t]| \leq \max\{|X[t-i]|, 0 \leq i < M\} \times \sum_{i=0}^{M-1} |C[i]| \quad (8.24)$$

$$|A[t]| \leq \sqrt{\sum_{i=0}^{M-1} |X[t-i]|^2} \times \sqrt{\sum_{i=0}^{M-1} |C[i]|^2} \quad (8.25)$$

ARM 嵌入式系统开发

式(8.24)说明,如果已知 $X[t]$ 的动态范围,那么动态范围的最大值以滤波系数 $C[i]$ 绝对值的总和为上限。式(8.25)说明,如果已知信号 $X[t]$ 的能量,那么 $A[t]$ 的动态范围以输入信号和系数能量的乘积为上限。这 2 个不等式都非常有用。对于给定的 $C[t]$,可以选择 $X[t]$ 使式子相等。它们是更通用的 Holder 不等式的特例,下面用例子来说明。

【例 8.8】考虑由下式(8.26)定义的简单、原生、高通滤波器。滤波器允许高频信号通过,但衰减低频信号。

$$y_t = -0.45x_t + 0.9x_{t-1} - 0.45x_{t-2} \quad (8.26)$$

假设使用 Q_n 和 Q_m 的 16 位定点信号 $X[t]$ 和 $C[i]$ 表示 x_t 和 c_i ,那么,

$$C[0] = -0.45 \times 2^m, C[1] = 0.90 \times 2^m, C[2] = -0.45 \times 2^m \quad (8.27)$$

既然 $X[t]$ 是一个 16 位整数, $|X[t]| \leq 2^{15}$, 因此,使用上面的第一个不等式,

$$|A[t]| \leq 2^{15} \times 1.8 \times 2^m = 1.8 \times 2^{15+m} \quad (8.28)$$

如果 $m \leq 15$, 那么 $A[t]$ 不会溢出 32 位整数。所以,用 $m=15$ 作最大系数精度。下面的整数计算用 16 位 Q_n 输入 $X[t]$ 和 32 位 $Q(n+15)$ 输出 $A[t]$, 实现滤波器:

$$A[t] = -0x399A * X[t] + 0x7333 * X[t-1] - 0x399A * X[t-2];$$

对于 Q_n 输出 $Y[t]$, 需要设 $Y[t] = A[t] \gg 15$, 然而,这样会溢出 16 位整数。因此这里需要饱和这个结果,或者用 $Q(n-1)$ 表示形式来保存结果。

块 FIR 滤波器

例 8.8 说明了通常可以使用乘积的整数和来实现滤波器,而无须检查饱和溢出:

$$A[t] = C[0] * X[t] + C[1] * X[t-1] + \dots + C[M-1] * X[t-M+1];$$

一般 $X[t]$ 和 $C[i]$ 是 k 位整数, $A[t]$ 是 $2k$ 位整数。这里 $k=8, 16$ 或 32 。表 8.3 列出了一些典型应用的精度要求。

表 8.3 不同应用的滤波器精度

应 用	$X[t]$ 的精度/位	$C[t]$ 的精度/位	$A[t]$ 的精度/位
视 频	8	8	16
电信音频	16	16	32
高质量音频	32	32	64

这里要介绍长 16 位和 32 位滤波器的详细例子。长滤波器的意思是 M 很大,以至于不

能把滤波器的系数保存在寄存器中。可以像例 8.8 那样在逐例的基础上优化短滤波器,短滤波器的系数可以放在寄存器中。

对于长滤波器,每个结果 $A[t]$ 依赖于从存储器读出的 M 个数据值和 M 个系数。这些读取比较耗时,并且仅仅计算单个结果 $A[t]$ 的效率很低。当装载数据和系数时,可以同时计算 $A[t+1]$ 甚至 $A[t+2]$ 。

一个 R 路(R -way)块滤波器使用一个单通路的数据 $X[t]$ 和系数 $C[i]$ 实现计算 R 个值 $A[t], A[t+1], \dots, A[t+R-1]$, 这样可以减少存储器访问的次数。相对于分别计算每个结果,只有 $1/R$ 访问次数,所以 R 应该尽可能大。另一方面, R 越大,就需要更多的寄存器来保存计算值、数据或系数。在实践中,只要在内循环中不会用尽寄存器,就可以尽可能选择大的 R 。在下面的例子中,将会说明在 ARM 上一般取 R 为 $2 \sim 6$ 。

一个 $R \times S$ 块滤波器是一个 R 路块滤波器。在这 R 路块滤波器中,对内循环的每个重复,一次读取 S 个数据和系数值。在每次循环中,累加 $R \times S$ 个乘积,获得 R 个累加值。

图 8.3 显示了一个典型的 4×3 块滤波器的实现。左边的每个累加值,是右边的系数和每列首行单个值的乘积的和。由于滤波程序将会按照存储器地址的增长顺序装载数据,图中从最早的采样点 X_{t-M+1} 开始。 4×3 滤波器的每个内循环累加了 4×3 平行四边形里的 12 个乘积。在图中把第 1 个平行四边形和第 3 个平行四边形的第一个采样点用阴影表示。

	X_{t-M+1}	X_{t-M+2}	X_{t-M+3}	X_{t-M+4}	X_{t-M+5}	X_{t-M+6}	X_{t-M+7}
A_t	C_{M-1}	C_{M-2}	C_{M-3}	C_{M-4}	C_{M-5}	C_{M-6}	C_{M-7}
A_{t+1}		C_{M-1}	C_{M-2}	C_{M-3}	C_{M-4}	C_{M-5}	C_{M-6}
A_{t+2}			C_{M-1}	C_{M-2}	C_{M-3}	C_{M-4}	C_{M-5}
A_{t+3}				C_{M-1}	C_{M-2}	C_{M-3}	C_{M-4}

图 8.3 一个 4×3 块滤波器的实现

从图 8.3 中可以看到,一个 $R \times S$ 块滤波器的实现需要 R 个累加值寄存器和 $R-1$ 个输入采样的历史记录,还需要一个寄存器来保存下一个系数。在把 S 个乘积加到每个累加器中后,循环开始重复,因此必须把 $X[t]$ 和 $X[t-S]$ 分配到同一个寄存器,同时必须把至少 $R-1$ 个采样的历史值保存在寄存器中,因此 $S \geq R-1$ 。基于这个原因,块滤波器通常需要 $R \times (R-1)$ 或 $R \times R$ 大小。

下面的例子给出了优化的块 FIR 实现。对于不同的 ARM 内核实现,要选择最佳的 R 和 S 值。

注意:在这些实现中,系数在存储器中反序存储。

ARM 嵌入式系统开发

图 8.3 显示了从系数 $C[M-1]$ 开始, 向后操作。

【例 8.9】如 ARM7TDMI 点乘一样, 这里保存 16 位和 32 位数据项在 32 位字中。这样可以使用多数据装载, 以获得最高装载效率。这个例子实现了一个 16 位输入数据的 4×3 块滤波器。数组指针 a, x 和 c 指向图 8.4 给出格式的输入和输出数组。

数组名	第一个元素	第二个元素	第三个元素	...	最后一个元素	数组长度
a	A_1	A_{n-1}	A_{n-2}	...	A_{n-M+1}	N
x	X_{n-M+1}	X_{n-M+2}	X_{n-M+3}	...	X_{n-1}	$N+M-1$
c	C_{M-1}	C_{M-2}	C_{M-3}	...	C_0	M

图 8.4 数组 a, x 和 c 的格式

注意数组 x 保存 $M-1$ 个采样的历史记录, 并且反置系数数组。这里把系数数组指针 c 和长度 M 放在一个结构中, 这样函数还可有 4 个寄存器参数。同时假设 N 是 4 的倍数, M 是 3 的倍数。

```

a      RN 0      ;输出采样数组 a[]
x      RN 1      ;输入采样数组 x[]
c      RN 2      ;系数数组 c[]
N      RN 3      ;输出的数目(4 的倍数)
M      RN 4      ;系数的数目(3 的倍数)
c_0    RN 3      ;系数寄存器
c_1    RN 12
c_2    RN 14
x_0    RN 5      ;数据寄存器
x_1    RN 6
x_2    RN 7
a_0    RN 8      ;输出累加器
a_1    RN 9
a_2    RN 10
a_3    RN 11

```

```

;void fir_16by16_arm7m
; (int *a,

```

```

;   int *x,
;   struct { int *c; unsigned int M; } *c,
;   unsigned int N)
fir_16by16_arm7m
    STMFD    sp!, {r4-r11, lr}
    LDMIA    c, {c, M}                ;装载系数数组和长度
next_sample_arm7m
    STMFD    sp!, {N, M}
    LDMIA    x!, {x_0, x_1, x_2}
    MOV      a_0, #0                  ;累加器清零
    MOV      a_1, #0
    MOV      a_2, #0
    MOV      a_3, #0
next_tap_arm7m
    ;处理下一个块(4x3=12)的 taps
    LDMIA    c!, {c_0, c_1, c_2}
    MLA      a_0, x_0, c_0, a_0
    MLA      a_0, x_1, c_1, a_0
    MLA      a_0, x_2, c_2, a_0
    MLA      a_1, x_1, c_0, a_1
    MLA      a_1, x_2, c_1, a_1
    MLA      a_2, x_2, c_0, a_2
    LDMIA    x!, {x_0, x_1, x_2}
    MLA      a_1, x_0, c_2, a_1
    MLA      a_2, x_0, c_1, a_2
    MLA      a_2, x_1, c_2, a_2
    MLA      a_3, x_0, c_0, a_3
    MLA      a_3, x_1, c_1, a_3
    MLA      a_3, x_2, c_2, a_3
    SUBS     M, M, #3                ;处理 3 个系数
    BGT      next_tap_arm7m
    LDMFD    sp!, {N, M}
    STMIA    a!, {a_0, a_1, a_2, a_3}
    SUB      c, c, M, LSL#2          ;恢复系数指针
    SUB      x, x, M, LSL#2          ;恢复数据指针
    ADD      x, x, # (4-3) * 4       ;提前数据指针
    SUBS     N, N, #4                ;4 个采样点滤波

```

```

    BGT      next_sample_arm7m
    LDMFD    sp!, {r4 - r11, pc}

```

每个内循环的重复操作处理接下去的 3 个系数,并更新 4 个滤波输出。假设系数 16 位,每个乘累加需要 4 个周期,那么在 62 个周期里,它处理了 12 个滤波器 tap(抽头)。块 FIR 的性能是每个 tap(抽头)平均 5.17 个周期(5.17 cycles/tap)。

注意:使用减法来复位系数和输入指针 c 和 x ,可减少开销,所以没有把它们值保存到堆栈中。

【例 8.10】 给出针对 ARM9TDMI 的一个优化的块滤波器。

首先,ARM9TDMI 有单周期 16 位装载指令,所以使用多数据装载没有多大好处,可以通过存储数据和系数在 16 位半字来节省存储器;其次,可使用 4×4 块滤波器实现,而不用 4×3 实现,这将减少外部的循环,并且当系数的数目是 4 的倍数而不是 3 的倍数时,用 4×4 块滤波器是很好的。

除了现在输入数组是 16 位,输入和输出数组和例 8.9 的格式一样。输出和系数的数目 N 和 M 都是 4 的倍数。

```

a      RN 0      ;输出采样数组 a[]
x      RN 1      ;输入采样数组 x[]
c      RN 2      ;系数数组 c[]
N      RN 3      ;输出的数目(4 的倍数)
M      RN 4      ;系数的数目(4 的倍数)
c_0    RN 3      ;系数寄存器
c_1    RN 12
x_0    RN 5      ;数据寄存器
x_1    RN 6
x_2    RN 7
x_3    RN 14
a_0    RN 8      ;输出累加器
a_1    RN 9
a_2    RN 10
a_3    RN 11

;void fir_16by16_arm9m
; (int *a,
;  short *x,
;  struct { short *c; unsigned int M; } *c,
;  unsigned int N)
fir_16by16_arm9m
    STMFD    sp!, {r4 - r11, lr}

```



```

        LDMIA    c, {c, M}
next_sample_arm9m
        STMED    sp!, {N, M}
        LDRSH    x_0, [x], #2
        LDRSH    x_1, [x], #2
        LDRSH    x_2, [x], #2
        LDRSH    x_3, [x], #2
        MOV      a_0, #0
        MOV      a_1, #0
        MOV      a_2, #0
        MOV      a_3, #0
next_tap_arm9m
        ;处理下一个块(4×4=16)的 taps
        LDRSH    c_0, [c], #2
        LDRSH    c_1, [c], #2
        SUBS     M, M, #4
        MLA      a_0, x_0, c_0, a_0
        LDRSH    x_0, [x], #2
        MLA      a_1, x_1, c_0, a_1
        MLA      a_2, x_2, c_0, a_2
        MLA      a_3, x_3, c_0, a_3
        LDRSH    c_0, [c], #2
        MLA      a_0, x_1, c_1, a_0
        LDRSH    x_1, [x], #2
        MLA      a_1, x_2, c_1, a_1
        MLA      a_2, x_3, c_1, a_2
        MLA      a_3, x_0, c_1, a_3
        LDRSH    c_1, [c], #2
        MLA      a_0, x_2, c_0, a_0
        LDRSH    x_2, [x], #2
        MLA      a_1, x_3, c_0, a_1
        MLA      a_2, x_0, c_0, a_2
        MLA      a_3, x_1, c_0, a_3
        MLA      a_0, x_3, c_1, a_0
        LDRSH    x_3, [x], #2
        MLA      a_1, x_0, c_1, a_1
        MLA      a_2, x_1, c_1, a_2
        MLA      a_3, x_2, c_1, a_3
        BGT      next_tap_arm9m

```

ARM 嵌入式系统开发

```

LDMFD    sp!, {N, M}
STMIA    a!, {a_0, a_1, a_2, a_3}
SUB      c, c, M, LSL#1          ;恢复系数指针
SUB      x, x, M, LSL#1          ;提前数据指针
SUBS     N, N, #4                ;4 个采样点滤波
BGT      next_sample_arm9m
LDMFD    sp!, {r4 - r11, pc}

```

这段代码已被调整,使得在接下去的 2 个周期不使用装载值。可以把循环计数器递减的代码移到循环开始的位置,以利用装载延时的间隙。

内循环的每个重复处理接下去的 4 个系数,并且更新 4 个滤波输出。假设系数是 16 位,每个乘累加需 4 个周期,因此它在 76 个周期里处理 16 个滤波 tap(抽头),得到 4.75 cycles/tap 的块滤波性能。

这段代码对于其它 ARMv4 体系结构的处理器,如 StrongARM,也可以很好地工作。在 StrongARM 上内循环需要 61 个周期,即 3.81 cycles/tap。

【例 8.11】 ARM9E 比其之前的 ARM 处理器有更快的乘法器。当 2 个 16 位值被打包在一个 32 位字中时,ARMv5TE 16 位乘法指令也要解开 16 位数据。因此可以在寄存器中存储更多的数据和系数,从而使用更少的装载指令。

本例为 ARMv5TE 处理器实现一个 6×6 块滤波器。为了获得最高速度,程序进行了优化,所以程序比较长。如果不需要太好的性能,可以使用 4×4 块实现来减小代码的尺寸。

除了输入数组现在是 16 位值,输入和输出数组和例 8.9 有同样的格式。输入和系数的数目, N 和 M , 必须是 6 的倍数。输入数组必须是 32 位对齐的,并且存储器系统是小端配置的。如果要编写针对大端存储器系统的程序,那么须用宏来代替 SMLAxy 指令,依据大小端改变 T 和 B 的设置。例如下面的宏: SMLA00 定义 SMLABB 或 SMLATT, 分别用于小端和大端的存储器系统。如果 b 和 c 以 16 位数组读取,那么 SMLA00 通常用 $c[0]$ 乘以 $b[0]$ 而无须考虑大小端。

```

MACRO
SMLA00 $a, $b, $c, $d
IF {ENDIAN} = "big"
    SMLATT $a, $b, $c, $d
ELSE
    SMLABB $a, $b, $c, $d
ENDIF
MEND

```

为了使例子简单,没有使用这样的宏。下面的例子只是工作在小端存储器系统。

```

a      RN 0      ;输出采样数组 a[]
x      RN 1      ;输入采样数组 x[] (32 位对齐)
c      RN 2      ;系数数组 c[] (32 位对齐)
N      RN 3      ;输出的数目(6 的倍数)
M      RN 4      ;系数的数目(6 的倍数)
c_10   RN 0      ;系数对
c_32   RN 3
x_10   RN 5      ;采样对
x_32   RN 6
x_54   RN 7
a_0    RN 8      ;输出累加器
a_1    RN 9
a_2    RN 10
a_3    RN 11
a_4    RN 12
a_5    RN 14

```

```

;void fir_16by16_arm9e
; (int * a,
;  short * x,
;  struct { short * c; unsigned int M; } * c,
;  unsigned int N)

```

```

fir_16by16_arm9e

```

```

STMFD    sp!, {r4 - r11, lr}
LDMIA    c, {c, M}

```

```

next_sample_arm9e

```

```

STMFD    sp!, {a, N, M}
LDMIA    x!, {x_10, x_32, x_54}      ;预装载 6 个采样
MOV      a_0, #0                      ;累加器清零
MOV      a_1, #0
MOV      a_2, #0
MOV      a_3, #0
MOV      a_4, #0
MOV      a_5, #0

```

```

next_tap_arm9e

```

```

;处理下一个块(6x6 = 36)的 taps
LDMIA    c!, {c_10, c_32}            ;装载 4 个系数
SUBS     M, M, #6

```

ARM 嵌入式系统开发

```

SMLABB    a_0, x_10, c_10, a_0
SMLATB    a_1, x_10, c_10, a_1
SMLABB    a_2, x_32, c_10, a_2
SMLATB    a_3, x_32, c_10, a_3
SMLABB    a_4, x_54, c_10, a_4
SMLATB    a_5, x_54, c_10, a_5
SMLATT    a_0, x_10, c_10, a_0
LDR        x_10, [x], #4           ;装载 2 个系数
SMLABT    a_1, x_32, c_10, a_1
SMLATT    a_2, x_32, c_10, a_2
SMLABT    a_3, x_54, c_10, a_3
SMLATT    a_4, x_54, c_10, a_4
SMLABT    a_5, x_10, c_10, a_5
LDR        c_10, [c], #4
SMLABB    a_0, x_32, c_32, a_0
SMLATB    a_1, x_32, c_32, a_1
SMLABB    a_2, x_54, c_32, a_2
SMLATB    a_3, x_54, c_32, a_3
SMLABB    a_4, x_10, c_32, a_4
SMLATB    a_5, x_10, c_32, a_5
SMLATT    a_0, x_32, c_32, a_0
LDR        x_32, [x], #4
SMLABT    a_1, x_54, c_32, a_1
SMLATT    a_2, x_54, c_32, a_2
SMLABT    a_3, x_10, c_32, a_3
SMLATT    a_4, x_10, c_32, a_4
SMLABT    a_5, x_32, c_32, a_5
SMLABB    a_0, x_54, c_10, a_0
SMLATB    a_1, x_54, c_10, a_1
SMLABB    a_2, x_10, c_10, a_2
SMLATB    a_3, x_10, c_10, a_3
SMLABB    a_4, x_32, c_10, a_4
SMLATB    a_5, x_32, c_10, a_5
SMLATT    a_0, x_54, c_10, a_0
LDR        x_54, [x], #4
SMLABT    a_1, x_10, c_10, a_1
SMLATT    a_2, x_10, c_10, a_2
SMLABT    a_3, x_32, c_10, a_3

```

```

SMLATT    a_4, x_32, c_10, a_4
SMLABT    a_5, x_54, c_10, a_5
BGT       next_tap_arm9e
LDMFD     sp!, {a, N, M}
STMIA     a!, {a_0, a_1, a_2, a_3, a_4, a_5}
SUB        c, c, M, LSL#1          ;恢复系数指针
SUB        x, x, M, LSL#1          ;提前数据指针
SUBS       N, N, #6
BGT       next_sample_arm9e
LDMFD     sp!, {r4 - r11, pc}

```

内循环的每次重复更新相邻的 6 个滤波输出,累加 6 个乘积到每个输出。表 8.4 显示了 ARMv5TE 体系结构处理器的运行周期定时。

表 8.4 ARMv5TE 16 位块滤波周期定时

处理器	内循环时钟周期	滤波器性能 cycles/tap
ARM9E	46	46/36=1.28
ARM10E	78	78/36=2.17
XScale	46	46/36=1.28

【例 8.12】 有时 16 位数据项不能提供足够的动态范围。ARMv5TE 体系结构增加一条指令 SMLAWx,使 32 位数据被 16 位系数滤波更有效。这条指令用一个 16 位系数乘以 32 位数据项,截取 48 位结果的高 32 位,并把它加到 32 位的累加器中。

这个例子实现了一个 32 位数据和 16 位系数的 5×4 块滤波器。除了系数数组是 16 位的外,输入和输出数组的格式和例 8.9 一样。输出的数目必须是 5 的倍数,系数的数目则是 4 的倍数。输入系数数组必须是 32 位对齐的,存储器系统是小端配置。像例 8.11 描述的一样,可以用宏定义编写大/小端无关的代码。

如果输入采样和系数分别使用 Q_n 和 Q_m 形式,那么输出使用 $Q(n+m-16)$ 。SMLAWx 右移 16 位,以防止溢出。

```

a          RN 0          ;输出采样数组 a[]
x          RN 1          ;输入采样数组 x[]
c          RN 2          ;系数数组 c[] (32 位对齐)
N          RN 3          ;输出的数目(5 的倍数)
M          RN 4          ;系数的数目(4 的倍数)
c_10       RN 0          ;系数对
c_32       RN 3
x_0        RN 5          ;输入采样

```

```

x_1      RN 6
x_2      RN 7
x_3      RN 14
a_0      RN 8      ;输出累加器
a_1      RN 9
a_2      RN 10
a_3      RN 11
a_4      RN 12

```

```

;void fir_32by16_arm9e
; (int *a,
;  int *x,
;  struct { short *c,unsigned int M;} *c,
;  unsigned int N)

fir_32by16_arm9e
    STMFD      sp!, {r4-r11, lr}
    LDMIA      c, {c, M}                      ;装载系数数组和长度

next_sample32_arm9e
    STMFD      sp!, {a, N, M}
    LDMIA      x!, {x_0, x_1, x_2, x_3}
    MOV        a_0, #0                        ;累加器清零
    MOV        a_1, #0
    MOV        a_2, #0
    MOV        a_3, #0
    MOV        a_4, #0

next_tap32_arm9e
    ;处理下一个块(5x4 = 20)的 taps
    LDMIA      c!, {c_10, c_32}
    SUBS       M, M, #4
    SMLAWB     a_0, x_0, c_10, a_0
    SMLAWB     a_1, x_1, c_10, a_1
    SMLAWB     a_2, x_2, c_10, a_2
    SMLAWB     a_3, x_3, c_10, a_3
    SMLAWT     a_0, x_1, c_10, a_0
    LDMIA      x!, {x_0, x_1}
    SMLAWT     a_1, x_2, c_10, a_1
    SMLAWT     a_2, x_3, c_10, a_2

```

```

SMLAWB    a_0, x_2, c_32, a_0
SMLAWB    a_1, x_3, c_32, a_1
SMLAWT    a_0, x_3, c_32, a_0
LDMIA     x!, {x_2, x_3}
SMLAWB    a_4, x_0, c_10, a_4
SMLAWT    a_3, x_0, c_10, a_3
SMLAWT    a_4, x_1, c_10, a_4
SMLAWB    a_2, x_0, c_32, a_2
SMLAWB    a_3, x_1, c_32, a_3
SMLAWB    a_4, x_2, c_32, a_4
SMLAWT    a_1, x_0, c_32, a_1
SMLAWT    a_2, x_1, c_32, a_2
SMLAWT    a_3, x_2, c_32, a_3
SMLAWT    a_4, x_3, c_32, a_4
BGT       next_tap32_arm9e
LDMFD     sp!, {a, N, M}
STMIA     a!, {a_0, a_1, a_2, a_3, a_4}
SUB        c, c, M, LSL#1
SUB        x, x, M, LSL#2
ADD        x, x, #(5-4)*4
SUBS       N, N, #5
BGT       next_sample32_arm9e
LDMFD     sp!, {r4-r11, pc}

```

内循环的每次重复更新 5 个滤波输出,累加 4 个乘积到每个输出。表 8.5 显示了 ARMv5TE 体系结构处理器的周期定时。

表 8.5 ARMv5TE 32×16 滤波周期定时

处理器	内循环时钟周期	滤波器性能 cycles/tap
ARM9E	30	30/20=1.5
ARM10E	44	44/20=2.2
XScale	34	34/20=1.7

【例 8.13】 高质量的音频应用通常需要比 16 位高的采样精度。在 ARM 上可以使用长乘指令 SMLAL 来实现一个 32 位输入数据和系数的高效的滤波器,输出值是 64 位。这使得 ARM 非常适合 CD 音质的应用。

输出和输入数组和例 8.9 有同样的数据格式。这里实现一个 3×2 的块滤波器,所以 N

必须是 3 的倍数, M 是 2 的倍数。在任何 ARMv4 实现的处理器上, 该滤波器都工作得很好。

a	RN 0	; 输出采样数组 a[]
x	RN 1	; 输入采样数组 x[]
c	RN 2	; 系数数组 c[]
N	RN 3	; 输出数组(3 的倍数)
M	RN 4	; 系数的数目(2 的倍数)
c_0	RN 3	; 系数寄存器
c_1	RN 12	
x_0	RN 5	; 数据寄存器
x_1	RN 6	
a_0l	RN 7	; 累加器(低 32 位)
a_0h	RN 8	; 累加器(高 32 位)
a_1l	RN 9	
a_1h	RN 10	
a_2l	RN 11	
a_2h	RN 14	

```

; void fir_32by32
; (long long * a,
;  int * x,
;  struct { int * c; unsigned int M; } * c,
;  unsigned int N)

```

```

fir_32by32

```

```

    STMFD    sp!, {r4 - r11, lr}
    LDMIA    c, {c, M}

```

```

next_sample32

```

```

    STMFD    sp!, {N, M}
    LDMIA    x!, {x_0, x_1}
    MOV      a_0l, #0
    MOV      a_0h, #0
    MOV      a_1l, #0
    MOV      a_1h, #0
    MOV      a_2l, #0
    MOV      a_2h, #0

```

```

next_tap32

```

```

; 执行下一个块(3 × 2 = 6)的 taps

```



```

LDMIA      c!, {c_0, c_1}
SMLAL      a_0l, a_0h, x_0, c_0
SMLAL      a_1l, a_1h, x_1, c_0
SMLAL      a_0l, a_0h, x_1, c_1
LDMIA      x!, {x_0, x_1}
SUBS       M, M, #2
SMLAL      a_2l, a_2h, x_0, c_0
SMLAL      a_1l, a_1h, x_0, c_1
SMLAL      a_2l, a_2h, x_1, c_1
BGT        next_tap32
LDMFD      sp!, {N, M}
STMIA      a!, {a_0l, a_0h, a_1l, a_1h, a_2l, a_2h}
SUB        c, c, M, LSL#2
SUB        x, x, M, LSL#2
ADD        x, x, #(3-2)*4
SUBS       N, N, #3
BGT        next_sample32
LDMFD      sp!, {r4-r11, pc}

```

内循环的每次重复处理接下去的 2 个系数并更新 3 个滤波输出。假设系数使用 32 位的全部范围,乘法不会提前终止。程序对大多数 ARM 的实现都是最优的,表 8.6 给出了在一些处理器上的运行周期数。

表 8.6 32×32 位滤波周期定时

处理器	内循环时钟周期	滤波器性能 cycles/tap
ARM7TDMI	54	$54/6=9$
ARM9TDMI	50	$50/6=8.3$
StrongARM	31	$31/6=5.2$
ARM9E	26	$26/6=4.3$
ARM10E	22	$22/6=3.7$
XScale	22	$22/6=3.7$

小结 在 ARM 上实现 FIR 滤波器

- 如果 FIR 系数的数目足够少,那么应在寄存器中存放系数和采样值。系数经常被重复使用,这样会节省所需的寄存器数目。
- 如果 FIR 滤波器长度很长,那么可使用一个尺寸为 $R \times (R-1)$ 或 $R \times R$ 的块滤波器算法,应选择 ARM 上可使用的 14 个通用寄存器所允许的最大的 R 。

- 应确保输入数组根据访问大小对齐。当使用 LDRD 时,是 64 位对齐的。应确保数组的长度是块大小的倍数。
- 应调整代码,以避免所有装载或乘法引入的互锁。

8.4 IIR 滤波

无限冲激响应(IIR)滤波器是一个线性依赖于有限数量输入采样和有限数量的先前滤波器输出的数字滤波器。换句话说,它结合了 FIR 滤波器和从先前滤波输出的反馈。从数学上讲,对于一些系数 b_i 和 a_j :

$$y_t = \sum_{i=0}^M b_i x_{t-i} - \sum_{j=1}^L a_j y_{t-j} \quad (8.29)$$

如果输入冲激信号 $x=(1,0,0,0,\dots)$,那么 y_t 可能永远振荡下去。这就是为什么它有无限冲激响应。然而,对一个稳定的滤波器, y_t 会衰减为 0。本节将讨论这种滤波器的有效实现。

用式(8.29)可直接计算输出信号 y_t 。在这种情况下,代码和 8.3 节中的 FIR 类似。但是,这种计算方法在数字上可能会不稳定。把滤波器分解为一系列二阶节(biquads)—— $M=L=2$ 的 IIR 滤波器(双二阶滤波器),通常会更精确、高效。

$$y_t = b_0 x_t + b_1 x_{t-1} + b_2 x_{t-2} - a_1 y_{t-1} - a_2 y_{t-2} \quad (8.30)$$

通过用一些二阶节重复对数据滤波,可实现任何 IIR 滤波器。为了说明这一点,可以使用 z 变换。这种变换与每个信号 x_t 关联,多项式 $x(z)$ 定义为:

$$x(z) = \sum_i x_i z^{-i} \quad (8.31)$$

如果把 IIR 等式变换为 z 坐标,可以得到:

$$(1 + a_1 z^{-1} + \dots + a_L z^{-L}) y(z) = (b_0 + b_1 z^{-1} + \dots + b_M z^{-M}) x(z) \quad (8.32)$$

等效地:

$$y(z) = H(z)x(z), \text{ 其中 } H(z) = \frac{b_0 + b_1 z^{-1} + \dots + b_M z^{-M}}{1 + a_1 z^{-1} + \dots + a_L z^{-1}} \quad (8.33)$$

接下来,把 $H(z)$ 作为 2 个 z^{-1} 多项式的比。可以把多项式分解为二次因子,然后把 $H(z)$ 表示为二项式比 $H_i(z)$ 的乘积,每个 $H_i(z)$ 表示一个二阶节。

所以,现在需要有效地实现二阶节。按其字面意义,要为一个二阶节计算 y_t ,需要当前

采样 x_i 和 4 个历史项 $x_{i-1}, x_{i-2}, y_{i-1}, y_{i-2}$ 。但是, 有一个技巧可把所需要的历史项或状态值从 4 个减少到 2 个。这里定义一个中间信号 s_i

$$s_i = x_i - a_1 s_{i-1} - a_2 s_{i-2} \quad (8.34)$$

那么

$$y_i = b_0 s_i + b_1 s_{i-1} + b_2 s_{i-2} \quad (8.35)$$

换句话说, 在滤波器的 FIR 部分之前, 先处理滤波器的反馈部分。等价地, 在处理 $H(z)$ 的分子之前处理分母。现在每个双二阶滤波器(biquad filter)只需 2 个状态变量 s_{i-1} 和 s_{i-2} 。

系数 b_0 控制二阶节的幅度, 当处理一系列二阶节时, 可以假设 $b_0 = 1$, 再最后使用一个乘法或移位来校正信号幅度。所以, 总而言之, 这里已经把 IIR 简化为一系列二阶节滤波。这些二阶节的形式为:

$$s_i = x_i - a_1 s_{i-1} - a_2 s_{i-2}, y_i = b_0 s_i + b_1 s_{i-1} + b_2 s_{i-2} \quad (8.36)$$

要实现每个二阶节, 需要在 ARM 的寄存器中保存 6 个值 $-a_1, -a_2, b_1, b_2, s_{i-1}, s_{i-2}$ 的定点表示。装载一个新的二阶节需要 6 次装载; 装载一个新的采样点只需 1 次装载。因此对于内循环来讲, 采样点上的循环比二阶节的循环更有效些。

对于块 IIR, 把输入信号 x_i 分成 N 个采样点的大帧。信号经过多次处理, 每次通过寄存器允许的尽可能多的二阶节滤波。典型地, 对于 ARMv4 处理器, 每次经过 1 个二阶节滤波处理; 对于 ARMv5TE 处理器, 经过 2 个二阶节处理。下面的例子给出了针对不同 ARM 处理器的 IIR 代码。

【例 8.14】 在 ARM7TDMI 上实现一个 1×2 的块 IIR 滤波器。

每次内循环对接下去的 2 个输入采样使用一个双二阶滤波器(biquad filter)。输入数组的格式由图 8.5 给出。

数组名	第一个元素	第二个元素	第三个元素	最后一个元素	数组长度	
x	X_i	X_{n1}	X_{n2}	...	X_{nN-1}	N
y	Y_i	Y_{n1}	Y_{n2}	...	Y_{nM-1}	N
b	B_0	B_1	B_2	...	B_{M-1}	M

图 8.5 数组 x, y 和 b 的格式

ARM 嵌入式系统开发

每个二阶节 B_k 是一个 6 个值 $(-a_1, -a_2, b_1, b_2, s_{t-1}, s_{t-1})$ 的列表。与前面 ARM7TDMI 上的实现一样, 这里把 16 位输入值存储在 32 位整数中, 这样可以使用多寄存器装载。这里把二阶节系数按 Q14 定点格式存储, 采样点数 N 必须是偶数。

```

y      RN 0      ;输出采样 y[]的地址
x      RN 1      ;输入采样 x[]的地址
b      RN 2      ;biquads 的地址
N      RN 3      ;被滤波的采样点数目 (2 的倍数)
M      RN 4      ;使用 biquads 的数目
x_0    RN 2      ;输入采样
x_1    RN 4
a_1    RN 6      ;biquad 系数 - a[1] (Q14 格式)
a_2    RN 7      ;biquad 系数 - a[2] (Q14 格式)
b_1    RN 8      ;biquad 系数 + b[1] (Q14 格式)
b_2    RN 9      ;biquad 系数 + b[2] (Q14 格式)
s_1    RN 10     ;s[t-1]、s[t-2] (交替)
s_2    RN 11     ;s[t-2]、s[t-1] (交替)
acc0   RN 12     ;累加器
acc1   RN 14

```

```

;typedef struct {
;    int a1,a2;    /* Q14 系数 - a[1], - a[2] */
;    int b1,b2;    /* Q14 系数 + b[1], + b[2] */
;    int s1,s2;    /* s[t-1], s[t-2] */
;} biquad;

;
;void iir_q14_arm7m
; (int * y,
;  int * x,
;  struct { biquad * b; unsigned int M; } * b,
;  unsigned int N);
iir_q14_arm7m
    STMFD    sp!, {r4-r11, lr}
    LDMIA    b, {b, M}
next_biquad_arm7m
    LDMIA    b!, {a_1, a_2, b_1, b_2, s_1, s_2}
    STMFD    sp!, {b, N, M}
next_sample_arm7m
    ;使用 2×1 块 IIR

```

```

LDMIA    x!, {x_0, x_1}
;把 biquad 应用于采样点 0 (x_0)
MUL      acc0, s_1, a_1
MLA      acc0, s_2, a_2, acc0
MUL      acc1, s_1, b_1
MLA      acc1, s_2, b_2, acc1
ADD      s_2, x_0, acc0, ASR #14
ADD      x_0, s_2, acc1, ASR #14
;把 biquad 应用于采样点 1 (x_1)
MUL      acc0, s_2, a_1
MLA      acc0, s_1, a_2, acc0
MUL      acc1, s_2, b_1
MLA      acc1, s_1, b_2, acc1
ADD      s_1, x_1, acc0, ASR #14
ADD      x_1, s_1, acc1, ASR #14
STMIA    y!, {x_0, x_1}
SUBS     N, N, #2
BGT      next_sample_arm7m
LDMFD    sp!, {b, N, M}
STMDB    b, {s_1, s_2}
SUB      y, y, N, LSL #2
MOV      x, y
SUBS     M, M, #1
BGT      next_biquad_arm7m
LDMFD    sp!, {r4 - r11, pc}

```

每个内循环在最坏情况下,将一个二阶节应用于 2 个采样点,需要 44 个时钟周期。所以对一般的二阶节,ARM7TDMI 给出的 IIR 性能是 22 个周期/二阶节-采样点(cycles/biquad-sample)。

【例 8.15】 在 ARM9TDMI 上可使用半字装载指令,而不用多寄存器装载。因此可以把采样点保存在 16 位的短整数中。这个例子实现了一个适合于 ARM9TDMI 的装载调度 IIR。除了这里使用 16 位数据项外,函数接口与例 8.14 相同。

y	RN 0	;输出采样 y[]的地址
x	RN 1	;输入采样 x[]的地址
b	RN 2	;biquads 的地址
N	RN 3	;被滤波的采样点数目(2 的倍数)
M	RN 4	;使用 biquads 的数目

ARM 嵌入式系统开发

```

x_0    RN 2      ;输入采样
x_1    RN 4
round  RN 5      ;舍入值 (1 << 13)
a_1    RN 6      ;biquad 系数 - a[1] (Q14 格式)
a_2    RN 7      ;biquad 系数 - a[2] (Q14 格式)
b_1    RN 8      ;biquad 系数 + b[1] (Q14 格式)
b_2    RN 9      ;biquad 系数 + b[2] (Q14 格式)
s_1    RN 10     ;s[t-1], s[t-2] (交替)
s_2    RN 11     ;s[t-2], s[t-1] (交替)
acc0   RN 12     ;累加器
acc1   RN 14

;typedef struct {
;    short a1,a2;      /* Q14 格式系数 - a[1], - a[2] */
;    short b1,b2;      /* Q14 格式系数 + b[1], + b[2] */
;    short s1,s2;      /* s[t-1], s[t-2] */
;} biquad;

;
;void iir_q14_arm9m
;    (short * y,
;     short * x,
;     struct { biquad * b; unsigned int M; } * b,
;     unsigned int N);

iir_q14_arm9m
    STMFD    sp!, {r4 - r11, lr}
    LDMIA    b, {b, M}
    MOV      round, #1 << 13

iir_next_biquad
    LDRSH    a_1, [b], #2
    LDRSH    a_2, [b], #2
    LDRSH    b_1, [b], #2
    LDRSH    b_2, [b], #2
    LDRSH    s_1, [b], #2
    LDRSH    s_2, [b], #2
    STMFD    sp!, {b, N, M}

iir_inner_loop
    ;使用 2x1 块 IIR
    ;把 biquad 应用于采样点 0 (x_0)
    MLA      acc0, s_1, a_1, round

```

```

LDRSH    x_0, [x], #2
MLA      acc0, s_2, a_2, acc0
MLA      acc1, s_1, b_1, round
MLA      acc1, s_2, b_2, acc1
ADD      s_2, x_0, acc0, ASR #14
ADD      x_0, s_2, acc1, ASR #14
STRH     x_0, [y], #2
;把 biquad 应用于 x_1
MLA      acc0, s_2, a_1, round
LDRSH    x_1, [x], #2
MLA      acc0, s_1, a_2, acc0
MLA      acc1, s_2, b_1, round
MLA      acc1, s_1, b_2, acc1
ADD      s_1, x_1, acc0, ASR #14
ADD      x_1, s_1, acc1, ASR #14
STRH     x_1, [y], #2
SUBS     N, N, #2
BGT      iir_inner_loop
LDMFD    sp!, {b, N, M}
STRH     s_1, [b, #-4]
STRH     s_2, [b, #-2]
SUB      y, y, N, LSL#1
MOV      x, y
SUBS     M, M, #1
BGT      iir_next_biquad
LDMFD    sp!, {r4-r11, pc}

```

表 8.7 显示了 ARM9TDMI 和 StrongARM 的周期定时。

表 8.7 ARMv4T IIR 周期定时

处理器	每个循环的周期数	每个 biquad-sample 的周期数
ARM9TDMI	44	22
StrongARM	33	16.5

【例 8.16】 对于 ARMv5TE 处理器,可以把 2 个 16 位值打包到一个寄存器中。这意味着可以同时把 2 个二阶节的状态和系数保存到寄存器中。这个例子实现了一个 2×2 的块 IIR 滤波器。内循环的每次重复将 2 个双二阶滤波器(biquads filter)应用于接下去的 2 个采样。

ARM 嵌入式系统开发

本例除使用 16 位的数组外,输入数组的格式和例 8.14 一样。二阶节的数组必须是 32 位对齐的。采样点的数目 N 和二阶节的数目 M 必须是偶数。

就像 ARM9E 的 FIR 一样,这个程序只能工作在小端存储器系统。如何用宏编写大/小端无关的 DSP 代码,可参考例 8.11 节的讨论。

```

y      RN 0      ;输出采样 y[]的地址
x      RN 1      ;输入采样 x[]的地址
b      RN 2      ;biquads 的地址 (32 位对齐)
N      RN 3      ;被滤波的采样点数目(2 的倍数)
M      RN 4      ;使用 biquads 的数目(2 的倍数)
x_0    RN 2      ;输入采样
x_1    RN 4
s_0    RN 5      ;新状态
b0_a21 RN 6      ;biquad 0, packed - a[2], - a[1]
b0_b21 RN 7      ;biquad 0, packed + b[2], + b[1]
b0_s_1 RN 8      ;biquad 0, s[t-1]
b0_s_2 RN 9      ;biquad 0, s[t-2]
b1_a21 RN 10     ;biquad 1, packed - a[2], - a[1]
b1_b21 RN 11     ;biquad 1, packed + b[2], + b[1]
b1_s_1 RN 12     ;biquad 1, s[t-1]
b1_s_2 RN 14     ;biquad 1, s[t-2]

```

```

;typedef struct {
;    short a1,a2;      /* Q14 格式系数 - a[1], - a[2] */
;    short b1,b2;      /* Q14 格式系数 + b[1], + b[2] */
;    short s1,s2;      /* s[t-1], s[t-2] */
;} biquad;
;
;void iir_q14_arm9e
; (short * y,
;  short * x,
;  struct { biquad * b,unsigned int M;} * b,
;  unsigned int N);
iir_q14_arm9e
    STMFED    sp!, {r4-r11, lr}
    LDMIA     b, {b, M}
next_biquad_arm9e
    LDMIA     b!, {b0_a21, b0_b21}
    LDRSH     b0_s_1, [b], #2

```



```

LDRSH    b0_s_2, [b], #2
LDMIA    b!, {b1_a21, b1_b21}
LDRSH    b1_s_1, [b], #2
LDRSH    b1_s_2, [b], #2
STMFD    sp!, {b, N, M}
next_sample_arm9e
;使用 2×2 块 IIR
LDRSH    x_0, [x], #2
LDRSH    x_1, [x], #2
SUBS     N, N, #2
MOV      x_0, x_0, LSL #14
MOV      x_1, x_1, LSL #14
;把 biquad 0 应用于采样点 0
SMLABB   x_0, b0_s_1, b0_a21, x_0
SMLABT   s_0, b0_s_2, b0_a21, x_0
SMLABB   x_0, b0_s_1, b0_b21, s_0
SMLABT   x_0, b0_s_2, b0_b21, x_0
MOV      b0_s_2, s_0, ASR #14
;把 biquad 0 应用于采样点 1
SMLABB   x_1, b0_s_2, b0_a21, x_1
SMLABT   s_0, b0_s_1, b0_a21, x_1
SMLABB   x_1, b0_s_2, b0_b21, s_0
SMLABT   x_1, b0_s_1, b0_b21, x_1
MOV      b0_s_1, s_0, ASR #14
;把 biquad 1 应用于采样点 0
SMLABB   x_0, b1_s_1, b1_a21, x_0
SMLABT   s_0, b1_s_2, b1_a21, x_0
SMLABB   x_0, b1_s_1, b1_b21, s_0
SMLABT   x_0, b1_s_2, b1_b21, x_0
MOV      b1_s_2, s_0, ASR #14
;把 biquad 1 应用于采样点 1
SMLABB   x_1, b1_s_2, b1_a21, x_1
SMLABT   s_0, b1_s_1, b1_a21, x_1
SMLABB   x_1, b1_s_2, b1_b21, s_0
SMLABT   x_1, b1_s_1, b1_b21, x_1
MOV      b1_s_1, s_0, ASR #14
MOV      x_0, x_0, ASR #14
MOV      x_1, x_1, ASR #14
STRH     x_0, [y], #2

```

```

STRH    x_1, [y], #2
BGT     next_sample_arm9e
LDMFD   sp!, {b, N, M}
STRH    b0_s_1, [b, #-12-4]
STRH    b0_s_2, [b, #-12-2]
STRH    b1_s_1, [b, #-4]
STRH    b1_s_2, [b, #-2]
SUB     y, y, N, LSL#1
MOV     x, y
SUBS    M, M, #2
BGT     next_biquad_arm9e
LDMFD   sp!, {r4-r11, pc}

```

表 8.8 显示了 ARM9E, ARM10E 和 XScale 的周期定时。

表 8.8 ARMv5E 的 IIR 周期定时

处理器	每个循环的周期数	每个 biquad-sample 的周期数
ARM9E	32	8.0
ARM10E	45	11.2
XScale	30	7.7

小结 实现 16 位 IIR 滤波器

- 把 IIR 分解为一系列二阶节(biquads)。选择数据精度,防止在 IIR 计算中溢出。为了计算 IIR 的最高增益,将一个脉冲作用于冲激 IIR,产生冲激响应。把 8.3 节的等式应用于冲激响应 $c[j]$ 。
- 使用块 IIR 算法,把要滤波的信号分成大的帧。
- 对采样帧每次用 M 个二阶节进行滤波。选择 M 为允许的二阶节的最大数目,即可以在 ARM 的 14 个寄存器中存放二阶节的状态和系数。应保证二阶节的总数是 M 的倍数。
- 和以往一样,要调整、优化代码,以避免装载和乘法引入互锁。

8.5 离散傅里叶变换

离散傅里叶变换 DFT(Discrete Fourier Transform)可以把一个时域信号 x_i 转换为频域信号 y_i 。其对应的逆变换(IDFT)可以从频域信号重建时域信号。这种方法在信号分析和压缩中被广泛使用。由于有一个算法——快速傅里叶变换 FFT(Fast Fourier Trans-

form, 可以非常高效地实现 DFT), 这种方法显得特别有用。本节将讨论 FFT 在一些 ARM 上的有效实现。

DFT 作用于长度是 N 的复数的时间采样序列, 将其转换为 N 个复数频率系数。这里用式(8.37)和(8.38)作为其定义。这些定义在不同书上会略有不同, 因为一些作者可能使用不同的缩减因子(scaling)或正变换和逆变换的不同定义。

$$y = \text{DFT}_N(x) \text{ 即 } y_k = \sum_{n=0}^{N-1} x_n w_N^{kn}, \quad w_N = e^{-2\pi i/N} \quad (8.37)$$

$$x = \text{IDFT}_N(y) \text{ 即 } x_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k w_N^{kn}, \quad w_N = e^{2\pi i/N} \quad (8.38)$$

可见, 除了缩减因子和 w_N 的选择外, 变换式是相同的。因此下面只考虑正向变换。事实上快速傅里叶变换算法适合于任何的 w_N , 只要 $w_N^N = 1$ 。这个算法只有在单位主极点(principal root of unity) $w_N^N \neq 1 (k < N)$ 时是可逆的。

快速傅里叶变换

FFT 的思想是, 用因子 N 将变换分解。假如 $N = R \times S$, 把输出分成大小为 R 的 S 块, 把输入分成大小为 S 的 R 块。换句话说:

$$k = nR + m \quad \text{对于 } n = 0, 1, \dots, S-1 \quad \text{和} \quad m = 0, 1, \dots, R-1 \quad (8.39)$$

$$t = rS + s \quad \text{对于 } r = 0, 1, \dots, R-1 \quad \text{和} \quad s = 0, 1, \dots, S-1 \quad (8.40)$$

那么:

$$y[nR + m] = \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} x[rS + s] w_N^{(nR+m)(rS+s)} \quad (8.41)$$

$$y[nR + m] = \sum_{s=0}^{S-1} w_S^s w_N^{nm} \left(\sum_{r=0}^{R-1} x[rS + s] w_R^{rm} \right) \quad (8.42)$$

式(8.42)把 N 点 DFT 减少为 S 个 R 点 DFTs 和 N 个与系数 w_N^{nm} 乘法的集合, 以及 R 个 S 点 DFTs 的集合。特别地, 如果依次设置

$$(u[sR], u[sR + 1], \dots, u[sR + R - 1]) = \text{DFT}_R(x[s], x[s + S], \dots, x[s + (R - 1)S]) \quad (8.43)$$

$$v[sR + m] = w_N^{nm} u[sR + m] \quad (8.44)$$

那么

$$y[nR + m] = \sum_{s=0}^{S-1} w_S^s v[sR + m]$$

所以

$$(y[m], y[R + m], \dots, y[(S - 1)R + m]) =$$

$$\text{DFT}_S(v[m], v[R+m], \dots, v[(S-1)R+m]) \quad (8.45)$$

在实践中,可以重复这个过程,以有效地计算 R 和 S 点的 DFTs。当 N 有很多小的因子时,这种方法工作得很好。最有用的情况是当 N 是 2 的幂时。

1. 基 2 快速傅里叶变换

假设 $N=2^a$ 。使 $R=2^{a-1}$, $S=2$, 对 DFT 进行分解。

$$\text{由于 } \text{DFT}_2(v[m], v[R+m]) = (v[m] + v[R+m], v[m] - v[R+m])$$

$$\text{可得 } y[m] = u[m] + w_N^m u[R+m] \text{ 和 } y[R+m] = u[m] - w_N^m u[R+m] \quad (8.46)$$

这 2 个操作称为基 2 蝶形时域抽取法(decimation-in-time radix-2 butterfly)。 N 点 DFT 减少为 2 个 R 点 DFTs,接着是 $N/2$ 个蝶形运算。针对 2 的每个因数,重复分解 $R=2^{a-2}$ 的过程。结果是一个包括 a 阶段的算法,每个阶段计算 $N/2$ 个蝶形运算。

注意: 当从 $x[rS+s]$ 计算 $u[sR+m]$ 时,数据的顺序必须被改变。如果用变换的顺序存储 $x[t]$,则可避免这种情况。对于基 2 的快速傅里叶变换,所有的蝶形运算都可以同址执行(在输入数组的原来存储位置),只要输入数组 $x[t]$ 以位反转(bit-reversed)的方式存放——把 $x[t]$ 存在 $x[s]$,索引 s 的 a 位是索引 t 的 a 位的倒序。

还有另外一种 FFT 分解方法。可以选择 $R=2$ 和 $S=2^{a-1}$,重复对第 2 个因子进行分解。这就是频域抽取法基 2 变换(decimation-in-frequency radix-2 transform)。对于频域抽取法基 2 变换,蝶形运算是:

$$y[2s] = x[s] + x[S+s] \text{ 和 } v[2s+1] = w_N^s (x[s] - x[S+s]) \quad (8.47)$$

从 ARM 的优化观点看,重要的区别在于复数乘法的位置。时域抽取法与 w_N^m 的乘法在加减法之前,频域抽取法与 w_N^s 的乘法在加减法之后。一个定点的乘法包括乘法和右移。在 ARM 的操作指令中,桶形移位器在加法和减法之前,所以 ARM 更适合时域抽取法。

由于基 4 FFT 可提供更好的性能,所以不详细讨论基 2 FFT 的编码。

2. 基 4 快速傅里叶变换

除了把 N 作为 4 的幂, $N=4^b$ 以外,基 4 FFT 和基 2 FFT 非常相似。这里使用时域抽取法分解, $R=4^{b-1}$ 和 $S=4$ 。那么基 4 蝶形运算是:

$$\begin{aligned} & (y[m], y[R+m], y[2R+m], y[3R+m]) = \\ & \text{DFT}_4(u[m], w_N^m u[R+m], w_N^{2m} u[2R+m], w_N^{3m} u[3R+m]) \end{aligned} \quad (8.48)$$

4 点 DFT 不需要任何复数乘法。因此,基 4 时域抽取法需要 $\frac{bN}{4}$ 个基 4 蝶形运算,每个蝶形运算中有 3 个复数乘法。基 2 算法需要 $\frac{2bN}{2}$ 个基 2 蝶形运算,每个蝶形运算中有 1 个

复数乘法。因此,基 4 算法节省了 25% 的乘法。

考虑基 8 变换有相当的诱惑力。然而,这只能节省一个很小百分比的乘法,获得的效益将会被额外的装载和存储消耗掉。ARM 没有太多的寄存器来有效地支持通用的基 8 蝶形运算。基 4 蝶形运算正好合适:能够节省大量的乘法,并且巧妙地利用 14 个可以得到的 ARM 寄存器。

为了有效地实现基 4 蝶形运算,这里使用一个基 2 FFT 来计算 DFT_4 。假设输入是位反转的,可以使用 8 个 ARM 寄存器同址计算一个 4 点 DFT。接下来的宏 C_FFT4 是 FFT 实现的重要部分,它处理 4 点 DFT,同时对输入缩减,这可以很好地使用 ARM 桶形移位器。为了防止溢出,这里也把结果除以 4。

```
x0_r    RN    4    ; 数据寄存器(实部)
x0_i    RN    5    ; 数据寄存器(虚部)
x1_r    RN    6
x1_i    RN    7
x2_r    RN    8
x2_i    RN    9
x3_r    RN   10
x3_i    RN   11
y3_r    RN   x3_i
y3_i    RN   x3_r
; 4 点复数 FFT
;
;(x0,x1,x2,y3) = DFT4(x0,x2 >> s,x1 >> s,x3 >> s)/4
;
;x0 = (x0 + (x2 >> s) + (x1 >> s) + (x3 >> s))/4
;x1 = (x0 - i * (x2 >> s) - (x1 >> s) + i * (x3 >> s))/4
;x2 = (x0 - (x2 >> s) + (x1 >> s) - (x3 >> s))/4
;y3 = (x0 + i * (x2 >> s) - (x1 >> s) - i * (x3 >> s))/4
;
MACRO
C_FFT4      $s
;(x2,x3) = (x2 + x3, x2 - x3)
ADD         x2_r, x2_r, x3_r
ADD         x2_i, x2_i, x3_i
SUB         x3_r, x2_r, x3_r, LSL#1
SUB         x3_i, x2_i, x3_i, LSL#1
;(x0,x1) = (x0 + (x1 >> s), x0 - (x1 >> s))/4
```

```

MOV        x0_r, x0_r, ASR#2
MOV        x0_i, x0_i, ASR#2
ADD        x0_r, x0_r, x1_r, ASR#(2+$s)
ADD        x0_i, x0_i, x1_i, ASR#(2+$s)
SUB        x1_r, x0_r, x1_r, ASR#(1+$s)
SUB        x1_i, x0_i, x1_i, ASR#(1+$s)
; (x0, x2) = (x0 + (x2 >> s)/4, x0 - (x2 >> s)/4)
ADD        x0_r, x0_r, x2_r, ASR#(2+$s)
ADD        x0_i, x0_i, x2_i, ASR#(2+$s)
SUB        x2_r, x0_r, x2_r, ASR#(1+$s)
SUB        x2_i, x0_i, x2_i, ASR#(1+$s)
; (x1, y3) = (x1 - i * (x3 >> s)/4, x1 + i * (x3 >> s)/4)
ADD        x1_r, x1_r, x3_i, ASR#(2+$s)
SUB        x1_i, x1_i, x3_r, ASR#(2+$s)
SUB        y3_r, x1_r, x3_i, ASR#(1+$s)
ADD        y3_i, x1_i, x3_r, ASR#(1+$s)
MEND

```

这里还要使用宏 C_LDR 和 C_STR 来装载和存储复数值。这可以简化例 8.17 和 8.18 中的 FFT 程序代码。

```

;复数装载 x = [a], a += offset
MACRO
C_LDR    $x, $a, $offset
LDRSH   $x_i, [$a, #2]
LDRSH   $x_r, [$a], $offset
MEND

;复数存储, [a] = x, a += offset
MACRO
C_STR    $x, $a, $offset
STRH    $x_i, [$a, #2]
STRH    $x_r, [$a], $offset
MEND

```

【例 8.17】 对任何 ARMv4 体系结构的处理器实现一个 16 位基 4 FFT。

这里假设序列点数为 $n=4^b$ 。如果 N 是 2 的奇数幂, 那么需要替换例程, 开始阶段要用一个基 2 或基 8 的过程, 而不直接用基 4 过程。

代码使用了一个技巧, 只用 3 个实数乘法就实现了一个复数乘法。如果 $a+ib$ 是一个

复数数据项, 并且 $c+is$ 是一个复数系数

$$\text{那么} \quad (a+ib)(c-is) = [(b-a)s + a(c+s)] + i[(b-a)s + b(c-s)] \quad (8.49)$$

$$(a+ib)(c+is) = [(a-b)s + a(c-s)] + i[(a-b)s + b(c+s)] \quad (8.50)$$

当 $c+is=e^{2\pi i/N}$ 时, 这些分别是做前向和反向基 4 蝶形运算变换所需要的复数乘法。给定输入 $c-s$, s , $c+s$, a 和 b , 可以使用减法、乘法和 2 个乘累加计算上面 2 个式子。在系数查找表中存储 $(c-s, s)$, 在需要时计算 $c+s$ 。前向和反向变换可以使用同样的表。

使用下面的代码在 ARMv4 体系结构上处理基 4 变换。序列长度 N 必须是 4 的幂。算法实际上计算了 $\text{DFT}_N(x)/N$, 额外的缩减因子 N 防止溢出。算法使用前面定义的宏 C_FFT4 和存-取的宏。

```

;复数共轭乘法 a=(xr+i*xi)*(cr-i*ci)
; x=xr + i*xi
; w=(cr-ci) + i*ci
MACRO
C_MUL9m $a, $x, $w
SUB    t1, $x._i, $x._r      ;(xi-xr)
MUL    t0, t1, $w._i         ;(xi-xr)*ci
ADD    t1, $w._r, $w._i, LSL#1 ;(cr+ci)
MLA    $a._i, $x._i, $w._r, t0 ;xi*cr-xr*ci
MLA    $a._r, $x._r, t1, t0   ;xr*cr+xi*ci
MEND

```

```

Y      RN 0    ;输出复数数组 y[]
c      RN 0    ;系数数组
x      RN 1    ;输入复数数组 x[]
N      RN 2    ;采样点数(2 的幂)
S      RN 2    ;块的数目
R      RN 3    ;每块中采样点的数目
x0_r   RN 4    ;数据寄存器(实部)
x0_i   RN 5    ;数据寄存器(虚部)
x1_r   RN 6
x1_i   RN 7
x2_r   RN 8
x2_i   RN 9
x3_r   RN 10
x3_i   RN 11

```

ARM 嵌入式系统开发

```

y3_r    RN x3_i
y3_i    RN x3_r
t0      RN 12    ;临时寄存器 (scratch register)
t1      RN 14

;void fft_16_arm9m(short * y, short * x, unsigned int N)
fft_16_arm9m
    STMED    sp!, {r4-r11, lr}
    MOV      t0, #0                                ;位反转计数器

first_stage_arm9m
    ;第一阶段装载和位反转
    ADD      t1,x, t0, LSL#2
    C_LDR    x0, t1, N
    C_LDR    x2, t1, N
    C_LDR    x1, t1, N
    C_LDR    x3, t1, N
    C_FFT4    0
    C_STR    x0, y, #4
    C_STR    x1, y, #4
    C_STR    x2, y, #4
    C_STR    y3, y, #4
    EOR      t0, t0, N, LSR#3                        ;第 3 位增加
    TST      t0, N, LSR#3                            ;从顶部
    BNE      first_stage_arm9m
    EOR      t0, t0, N, LSR#4                        ;第 4 位增加
    TST      t0, N, LSR#4                            ;从顶部
    BNE      first_stage_arm9m
    MOV      t1, N, LSR#5                            ;第 5 位增加

bit_reversed_count_arm9m                        ;位递减 bits downward
    EOR      t0, t0, t1
    TST      t0, t1
    BNE      first_stage_arm9m
    MOVS     t1, t1, LSR#1
    BNE      bit_reversed_count_arm9m
    ;完成第一阶段
    SUB      x, y, N, LSL#2                        ;x = 工作缓冲
    MOV      R, #16

```



```

        MOVS    S, N, LSR#4
        LDMEQFD sp!, {r4 - r11, pc}
        ADR     c, fft_table_arm9m
next_stage_arm9m
        ;S = 块的数目
        ;R = 每块的采样点数目
        STMFD   sp!, {x, S}
        ADD     t0, R, R, LSL#1
        ADD     x, x, t0
        SUB     S, S, #1 << 16
next_block_arm9m
        ADD     S, S, R, LSL#(16 - 2)
next_butterfly_arm9m
        ;S = ((number butterflies left - 1) << 16)
        ;      + (number of blocks left)
        C_LDR   x0, x, -R
        C_LDR   x3, c, #4
        C_MUL9m x3, x0, x3
        C_LDR   x0, x, -R
        C_LDR   x2, c, #4
        C_MUL9m x2, x0, x2
        C_LDR   x0, x, -R
        C_LDR   x1, c, #4
        C_MUL9m x1, x0, x1
        C_LDR   x0, x, #0
        C_FFT4  14
        ;系数是 Q14 格式
        C_STR   x0, x, R
        C_STR   x1, x, R
        C_STR   x2, x, R
        C_STR   y3, x, #4
        SUBS    S, S, #1 << 16
        BGE     next_butterfly_arm9m
        ADD     t0, R, R, LSL#1
        ADD     x, x, t0
        SUB     S, S, #1
        MOVS    t1, S, LSL#16
        SUBNE   c, c, t0

```

ARM 嵌入式系统开发

```

BNE    next_block_arm9m
LDMFD  sp!, {x, S}
MOV     R, R, LSL#2                      ;块尺寸的4倍(quadrouple block size)
MOVS    S, S, LSR#2                      ;quarter number of blocks
BNE     next_stage_arm9m
LDMFD  sp!, {r4 ~ r11, pc}

```

fft_table_arm9m

```

;FFT twiddle table of triplets E(3t), E(t), E(2t)
;Where E(t) = (cos(t) - sin(t)) + i * sin(t) at Q14
;N = 16 t = 2 * PI * k/N for k = 0,1,2,...,N/4 - 1
DCW 0x4000,0x0000, 0x4000,0x0000, 0x4000,0x0000
DCW 0xdd5d,0x3b21, 0x22a3,0x187e, 0x0000,0x2d41
DCW 0xa57e,0x2d41, 0x0000,0x2d41, 0xc000,0x4000
DCW 0xdd5d,0xe782, 0xdd5d,0x3b21, 0xa57e,0x2d41
;N = 64 t = 2 * PI * k/N for k = 0,1,2,...,N/4 - 1
DCW 0x4000,0x0000, 0x4000,0x0000, 0x4000,0x0000
DCW 0x2aaa,0x1294, 0x396b,0x0646, 0x3249,0x0c7c
DCW 0x11a8,0x238e, 0x3249,0x0c7c, 0x22a3,0x187e
DCW 0xf721,0x3179, 0x2aaa,0x1294, 0x11a8,0x238e
DCW 0xdd5d,0x3b21, 0x22a3,0x187e, 0x0000,0x2d41
DCW 0xc695,0x3fb1, 0x1a46,0x1e2b, 0xee58,0x3537
DCW 0xb4be,0x3ec5, 0x11a8,0x238e, 0xdd5d,0x3b21
DCW 0xa963,0x3871, 0x08df,0x289a, 0xcdb7,0x3ec5
DCW 0xa57e,0x2d41, 0x0000,0x2d41, 0xc000,0x4000
DCW 0xa963,0x1e2b, 0xf721,0x3179, 0xb4be,0x3ec5
DCW 0xb4be,0x0c7c, 0xee58,0x3537, 0xac61,0x3b21
DCW 0xc695,0xf9ba, 0xe5ba,0x3871, 0xa73b,0x3537
DCW 0xdd5d,0xe782, 0xdd5d,0x3b21, 0xa57e,0x2d41
DCW 0xf721,0xd766, 0xd556,0x3d3f, 0xa73b,0x238e
DCW 0x11a8,0xcac9, 0xcdb7,0x3ec5, 0xac61,0x187e
DCW 0x2aaa,0xc2c1, 0xc695,0x3fb1, 0xb4be,0x0c7c
;N = 256 t = 2 * PI * k/N k = 0,1,2,...,N/4 - 1
;如果需要可以继续

```

上面的代码分 2 部分。第一阶段不需要任何复数运算,以位反转的顺序从源数组 x 中读取数据,接着使用基 4 蝶形运算,并写到目标数组 y ;在目标缓冲的原址处理剩余的阶段。在每个阶段交替源和目标缓冲,不使用位反转实现 FFT 是可行的。然而,在一般阶段

循环中这需要更多的寄存器,而这些寄存器是无法得到的。位反转的开销是非常小的,总的说来,每个输入采样点的开销小于 1.5 个周期。

前面代码的基准测试结果可参考 8.6 节。

【例 8.18】 实现一个在 ARMv5TE 体系结构处理器(如 ARM9E)的基 4 FFT。

对于 ARM9E,不需要利用例 8.17 中的技巧来把一个复数乘法简化为 3 个实数乘法。用正常的方法,使用单周期 16×16 位的乘法实现复数乘法更快。这也意味着可以使用一个 (c,s) 值的 Q15 系数表,而且变换要比例 8.17 有稍高的精度。这里省略了寄存器的分配,因为它与例 8.17 是一样的。

```

;复数共轭乘法 a = (xr + i * xi) * (cr - i * ci)
; x = xr + i * xi (封装在 32 位中的 2 个 16 位)
; w = cr + i * ci (封装在 32 位中的 2 个 16 位)
MACRO
C_MUL9e $a, $x, $w
SMULBT t0, $x, $w ;xr * ci
SMULBB $a._r, $x, $w ;xr * cr
SMULTB $a._i, $x, $w ;xi * cr
SMLATB $a._r, $x, $w, $a._r ;xr * cr + xi * ci
SUB $a._i, $a._i, t0 ;xi * cr - xr * ci
MEND

;void fft_16_arm9e(short *y, short *x, unsigned int N)
fft_16_arm9e
    STMFID sp!, {r4-r11, lr}
    MOV t0, #0 ;位反转计数器
    MVN R, #0x80000000 ;R = 0x7FFFFFFF

first_stage_arm9e
    ;第一阶段装载和位反转
    ADDS t1, x, t0, LSL#2 ;t1 = &x[t0] 并消除进位
    C_LDR x0, t1, N
    C_LDR x2, t1, N
    C_LDR x1, t1, N
    C_LDR x3, t1, N
    C_FFT4 0
    C_STR x0, y, #4
    C_STR x1, y, #4
    C_STR x2, y, #4

```

ARM 嵌入式系统开发

```

C_STR    y3, y, #4
;位反转增加模块(N/4)
RSC      t0, t0, N, LSR #2          ;t0 = (N/4) - t0 - 1
CLZ      t1, t0                    ;查找前导1(find leading 1)
EORS     t0, t0, R, ASR t1         ;“异或”前导1以外的位
BNE      first_stage_arm9e        ;如果计数不为0,则循环
;完成第一阶段
SUB      x, y, N, LSL #2          ;x = 工作缓冲
MOV      R, #16
MOVS     S, N, LSR #4
LDMEQFD sp!, {r4 - r11, pc}
ADR      c, fft_table_arm9e

next_stage_arm9e
;S = 块的数目
;R = 每块采样点的数目
STMED    sp!, {x, S}
ADD      t0, R, R, LSL #1
ADD      x, x, t0
SUB      S, S, #1 << 16

next_block_arm9e
ADD      S, S, R, LSL # (16 - 2)

next_butterfly_arm9e
;S = ((剩余蝶形运算数 - 1) << 16)
;    + 剩余块数)
LDR      x2_r, [x], -R            ;数据打包
LDR      x2_i, [c], #4            ;系数打包
LDR      x1_r, [x], -R
LDR      x1_i, [c], #4
LDR      x0_r, [x], -R
LDR      x0_i, [c], #4
C_MUL9e x3, x2_r, x2_i
C_MUL9e x2, x1_r, x1_i
C_MUL9e x1, x0_r, x0_i
C_LDR    x0, x, #0
C_FFT4   15                      ;系数是 Q15 格式
C_STR    x0, x, R
C_STR    x1, x, R

```

```

C_STR    x2, x, R
C_STR    y3, x, #4
SUBS     S, S, #1 << 16
BGE      next_butterfly_arm9e
ADD      t0, R, R, LSL #1
ADD      x, x, t0
SUB      S, S, #1
MOVS     t1, S, LSL #16
SUBNE    c, c, t0
BNE      next_block_arm9e
LDMFD    sp!, {x, S}
MOV      R, R, LSL #2           ;块尺寸的 4 倍(quadrouple block size)
MOVS     S, S, LSR #2           ;块数的 1/4
BNE      next_stage_arm9e
LDMFD    sp!, {r4 - r11, pc}

```

fft_table_arm9e

;FFT 数据表。E(3t), E(t), E(2t)

;E(t) = cos(t) + i * sin(t) 格式位 Q15

;N = 16 t = 2 * PI * k/N (k = 0, 1, 2, ..., N/4 - 1)

DCW 0x7fff, 0x0000, 0x7fff, 0x0000, 0x7fff, 0x0000

DCW 0x30fc, 0x7642, 0x7642, 0x30fc, 0x5a82, 0x5a82

DCW 0xa57e, 0x5a82, 0x5a82, 0x5a82, 0x0000, 0x7fff

DCW 0x89be, 0xcf04, 0x30fc, 0x7642, 0xa57e, 0x5a82

;N = 64 t = 2 * PI * k/N (k = 0, 1, 2, ..., N/4 - 1)

DCW 0x7fff, 0x0000, 0x7fff, 0x0000, 0x7fff, 0x0000

DCW 0x7a7d, 0x2528, 0x7f62, 0x0c8c, 0x7d8a, 0x18f9

DCW 0x6a6e, 0x471d, 0x7d8a, 0x18f9, 0x7642, 0x30fc

DCW 0x5134, 0x62f2, 0x7a7d, 0x2528, 0x6a6e, 0x471d

DCW 0x30fc, 0x7642, 0x7642, 0x30fc, 0x5a82, 0x5a82

DCW 0x0c8c, 0x7f62, 0x70e3, 0x3c57, 0x471d, 0x6a6e

DCW 0xe707, 0x7d8a, 0x6a6e, 0x471d, 0x30fc, 0x7642

DCW 0xc3a9, 0x70e3, 0x62f2, 0x5134, 0x18f9, 0x7d8a

DCW 0xa57e, 0x5a82, 0x5a82, 0x5a82, 0x0000, 0x7fff

DCW 0x8f1d, 0x3c57, 0x5134, 0x62f2, 0xe707, 0x7d8a

DCW 0x8276, 0x18f9, 0x471d, 0x6a6e, 0xcf04, 0x7642

DCW 0x809e, 0xf374, 0x3c57, 0x70e3, 0xb8e3, 0x6a6e

```

DCW 0x89be,0xcf04, 0x30fc,0x7642, 0xa57e,0x5a82
DCW 0x9d0e,0xaecc, 0x2528,0x7a7d, 0x9592,0x471d
DCW 0xb8e3,0x9592, 0x18f9,0x7d8a, 0x89be,0x30fc
DCW 0xdad8,0x8583, 0x0c8c,0x7f62, 0x8276,0x18f9
;N=256 t=2*PI*k/N (k=0,1,2,...,N/4-1)
;...如果需要可以继续

```

再次说明,这种方法实际上计算 $DFT_N(x)/N$,所以不会有溢出的可能。上面代码的基准测试结果可参考 8.6 节。

注意:在 ARMv5E 上使用 CLZ 指令来加速位反转中需要的计数操作。

小结 DFT 的实现

- 使用基 4、时域抽取法的 FFT 实现。如果序列点数不是 4 的幂,那么在第一阶段使用基 2 或基 8。
- 在算法开始,当为第一阶段读数据时,执行位反转。尽管可以不用位反转执行 FFT,但那样通常会在内循环中需要更多的寄存器。
- 如果一个标量的乘法需要超过 1 个周期,那么使用类似例 8.17 的方法把一个复数乘法分解成 3 个标量乘法。这是在 ARM9TDMI 上做 16 位 FFT 或在 ARM9E 上做 32 位 FFT 的情况。
- 为了防止溢出,在每次基 k 阶段,用 k 做缩减。或者确定 N 点 DFT 的输入有 N 倍的增长空间。这通常是实现 32 位 FFT 的情况。

8.6 总结

表 8.9 和表 8.10 总结了本章所介绍的程序的基准性能指数(benchmarks)。一般可以通过展开(unrolling)进一步调整这些代码,或为特殊应用编程。然而,这些数据将提供一个在实践中 ARM 系统获得固定的 DSP 性能的有用参考。这些基准性能指数包括所有的装载、存储和循环,并假定是 0 等待存储器或在有 cache 并命中的情况。

本章讨论了在 ARM 上有效实现定点 DSP 算法的途径,详细研究了 4 种基本的算法:点乘、块 FIR 滤波器、块 IIR 滤波器和离散傅里叶变换。同时考虑了在不同 ARM 体系结构和实现中的区别。

表 8.9 ARM 滤波基准性能指数

处理器核	16 位点乘 (cycles/tap)	16 位块 FIR 滤波器 (cycles/tap)	32 位块 FIR 滤波器 (cycles/tap)	16 位块 IIR 滤波器 (cycles/biquad)
ARM7TDMI	7.6	5.2	9.0	22.0
ARM9TDMI	7.0	4.8	8.3	22.0
StrongARM	4.8	3.8	5.2	16.5
ARM9E	2.5	1.3	4.3	8.0
Xscale	1.8	1.3	3.7	7.7

表 8.10 ARM FFT 基准性能指数

16 位复数 FFT(基 4)	ARM9TDMI(cycles/FFT)	ARM9E(cycles/FFT)
64 点	3 524	2 480
256 点	19 514	13 194
1 024 点	99 946	66 196
4 096 点	487 632	318 878

第 9 章

异常和中断处理

- 异常处理
- 中 断
- 中断处理方法
- 总 结

异常和中断处理是嵌入式系统的重要核心部分。它们负责处理错误、中断和其它由外部系统触发的事件。高效的异常处理能够大大改善系统的性能；同时，确定一个好的处理方法的过程，也是复杂，充满挑战而有乐趣的。

本章将介绍异常处理的理论与实践，特别是 ARM 处理器对中断的处理。ARM 处理器有 7 种可以使正常指令顺序中止执行的异常情况：数据中止、快速中断请求、中断请求、预取指中止、软件中断、复位及未定义指令。

本章分为 3 个主要部分：

- **异常处理** 包括 ARM 处理器处理异常的一些特定细节；
- **中断** ARM 把中断定义为一类特殊的异常，本节讨论了中断请求的使用，并介绍了一些有关中断处理的常用术语、特征和机制；
- **中断处理方法** 最后一节提供了一整套中断处理方法，包括对应每种方法的实现例子。

9.1 异常处理

异常是需要中止指令正常执行的任何情形，例如 ARM 内核产生复位，取指或存储器访问失败，遇到未定义指令，执行了软件中断指令，或者出现了一个外部中断等。异常处理就是处理这些异常情况的方法。

大多数异常都对应一个软件的异常处理程序——一个在异常发生时执行的软件程序。例如：一个数据中止异常就有一个数据中止处理程序。这个处理程序首先确定异常产生的原因，然后为该异常提供特定的服务。服务可以发生在处理程序内部，也可以跳转到一个专门的服务程序。复位异常是一个特例，它用来初始化一个嵌入式系统。

本节将介绍以下的异常处理主题：

- ARM 处理器模式及异常；
- 向量表；
- 异常的优先级；
- 链接寄存器偏移。

9.1.1 ARM 处理器模式及异常

表 9.1 列出了 ARM 处理器的各种异常。每种异常都导致内核进入一种特定的模式。此外，可以通过编程改变 cpsr，进入任何 ARM 处理器模式。用户和系统模式是仅有的可不

ARM 嵌入式系统开发

通过相应异常进入的 2 种模式,换句话说,要进入这 2 种模式,必须修改 cpsr。

当一个异常导致模式的改变时,内核自动地:

- 把 cpsr 保存到相应异常模式下的 spsr;
- 把 pc 保存到相应异常模式下的 lr;
- 设置 cpsr 为相应异常模式;
- 设置 pc 为相应异常处理程序的入口地址。

表 9.1 ARM 处理器异常及其对应的模式

异 常	模 式	主要目的
快速中断请求	FIQ	快速中断请求处理
中断请求	IRQ	中断请求处理
SWI 和复位	SVC	操作系统的受保护模式
预取指中止和数据中止	abort	虚存和(或)存储器保护处理
未定义指令	undefined	软件模拟硬件协处理器

图 9.1 给出了一个异常及其相应模式的简化示意图。

注意: 当一个异常发生时,ARM 处理器总是切换到 ARM 状态(即非 Thumb 状态)。

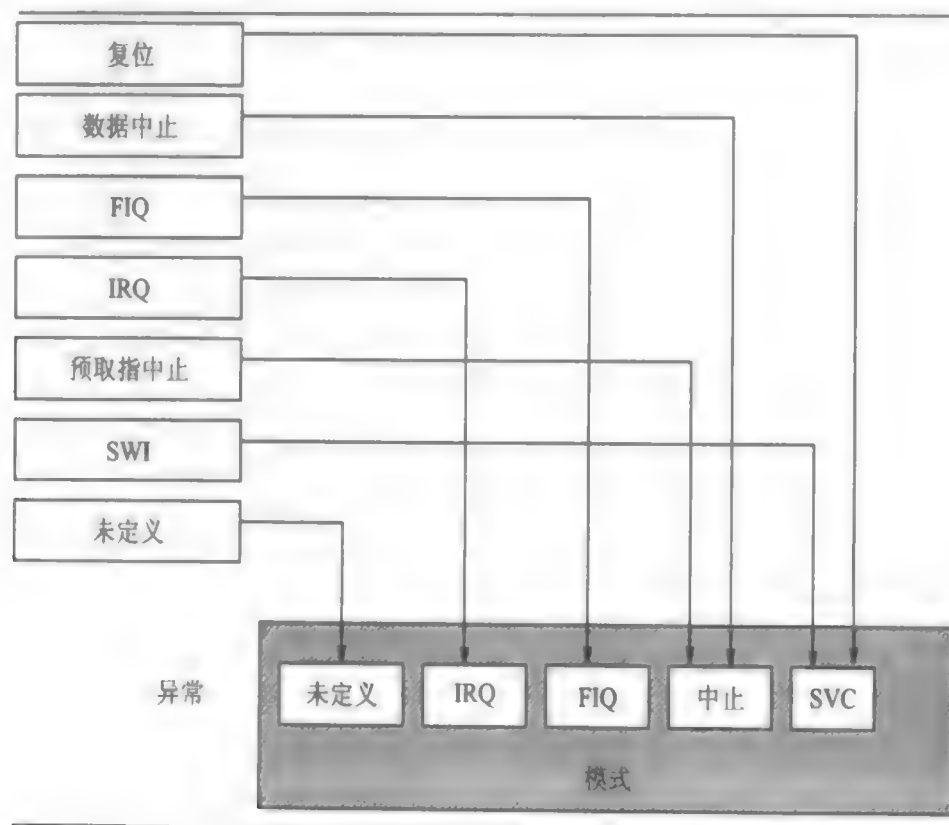


图 9.1 异常及其对应的模式

9.1.2 向量表

第2章介绍了向量表——异常发生时,ARM内核跳转地址组成的表。这些地址通常包含以下形式的跳转指令。

- B<address>——这条分支指令实现了相对于pc的分支跳转
- LDR pc,[pc,#offset]——这条寄存器装载指令把处理程序(handler)的入口地址从存储器装载到pc。该地址是一个32位的绝对地址,它储存在向量表附近。由于有额外的存储器访问,装载这4字节的绝对地址会使分支跳转到特定处理程序稍有延迟。不过,可以用这种方法,跳转到存储空间内的任意地址。
- LDR pc,[pc,#-0xff0]——这条寄存器装载指令把一个特殊的中断服务程序地址从地址0xfffff030装载到pc。只有当向量中断控制器存在时(VIC PL190),才能使用这条特殊的指令。
- MOV pc,#immediate——这条move指令把一个立即数复制到pc。它可跨越全部的地址空间,但要注意受到地址对齐问题的限制。这个地址必须是一个由8位立即数循环右移偶数次得到的。

也可以在向量表中使用其它类型的指令。例如FIQ处理程序可以从地址偏移+0x1c处开始,这样FIQ处理程序就可以不用跳转,立即从FIQ向量地址处开始执行,因为它位于向量表的最后。跳转指令使pc跳转到一个特定地址,以便处理某个特定的异常。

表9.2列出了每种异常所对应的模式和向量表偏移量。

表 9.2 向量表及处理器模式

异 常	模 式	向量表偏移
复位	SVC	+0x00
未定义指令	UND	+0x04
软件中断(SWI)	SVC	+0x08
预取指中止	ABT	+0x0c
数据中止	ABT	+0x10
未分配	—	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

【例9.1】图9.2给出了一个典型的向量表。未定义指令(异常)的入口地址是一条跳转到未定义处理程序的指令。其它向量使用指令LDR装载pc,以实现间接地址跳转。

ARM 嵌入式系统开发

注意：FIQ 处理程序也使用指令 LDR 把地址装载到 pc，而并没有利用它特殊位置（向量表最后一个）的好处——处理程序的入口可以直接放在 FIQ 向量入口地址处。

0x00000000	: 0xe59ffa38	RESET	:	> ldr pc, [pc, #reset]
0x00000004	: 0xea000502	UNDEF	:	b undInstr
0x00000008	: 0xe59ffa38	SWI	:	ldr pc, [pc, #swi]
0x0000000c	: 0xe59ffa38	PABT	:	ldr pc, [pc, #prefetch]
0x00000010	: 0xe59ffa38	DABT	:	ldr pc, [pc, #data]
0x00000014	: 0xe59ffa38	—	:	ldr pc, [pc, #notassigned]
0x00000018	: 0xe59ffa38	IRQ	:	ldr pc, [pc, #irq]
0x0000001c	: 0xe59ffa38	FIQ	:	ldr pc, [pc, #fiq]

图 9.2 向量表举例

9.1.3 异常优先级

异常可以同时发生，因此处理器必须采取一种基于优先级的机制。表 9.3 列出了 ARM 处理器的各种异常及其对应的优先级。例如，复位异常的优先级最高，处理器上电时发生复位异常。所以，当产生复位时，它将优先于其它异常得到处理。同样，当一个数据中止发生时，它将优先于除复位异常外的其它所有异常。优先级最低的 2 种异常是：软件中断和未定义指令异常。可以通过设置 cpsr 中的 I 位或 F 位来禁止某些异常，如表 9.3 所列。

表 9.3 异常优先级

异常	优先级	I 位	F 位
复位	1	1	1
数据中止	2	1	—
快速中断请求	3	1	1
中断请求	4	1	—
预取指中止	5	1	—
软件中断	6	1	—
未定义指令	6	1	—

每一种异常将按照表 9.3 中设置的优先级得到处理。下面从最高优先级异常开始，逐一介绍这些异常是如何被处理的。

复位异常是优先级最高的异常，一旦复位信号产生，总是会发生复位异常。复位异常处理程序对系统进行初始化，包括配置存储器和 cache。外部中断源必须在 IRQ 或 FIQ 中断允许之前初始化，以避免在还没有设置好相应的处理程序前产生中断。复位处理程序还要为所有处理器模式设置堆栈指针。

在执行复位处理程序的开头几句指令时，假设不会有别的异常或中断发生。编程时应避免 SWI、未定义指令及存储器访问导致的中止，即处理程序应仔细实现，以避免其它异常

的再次触发。

数据中止异常发生在存储控制器或 MMU 指示访问了无效的存储器地址时(例如对于给定的一个地址,没有对应的物理存储器存在),或者当前代码在没有正确的访问权限时,试图读/写存储器。由于没有禁止 FIQ 异常,在一个数据中止处理程序中,可以发生 FIQ 异常。当 FIQ 服务完成后,控制权交还给数据中止处理程序。

快速中断请求(FIQ)异常发生在一个外部设备把内核的 FIQ 线^{*}置为 nFIQ 时。FIQ 异常是优先级最高的中断。内核在进入 FIQ 处理程序时,把 FIQ 和 IRQ 都禁止了,因此任何外部中断源都不能再次中断处理器,除非在软件中重新允许了 IRQ 和(或)FIQ。应该仔细设计 FIQ 处理程序(对于数据中止,SWI 和 IRQ 也一样),以便高效地为异常处理服务。

中断请求(IRQ)异常发生在一个外部设备把内核的 IRQ 线置为 nIRQ 时。IRQ 异常是第二优先级的中断。FIQ 异常和数据中止异常都没有发生时,IRQ 处理程序才能够进入。在进入 IRQ 处理程序时,内核禁止 IRQ 异常,直到当前中断源被清除。

预取指中止异常即试图取指令而导致存储器访问失败的情形。在流水线中,如果某条指令(试图取的指令)的“执行”阶段没有优先级更高的异常出现,将发生预取指中止异常。在进入相应的处理程序时,内核禁止 IRQ 异常,而保持 FIQ 不变。如果允许了 FIQ,并且发生了一个 FIQ 异常,则它可在处理预取指中止过程中得到响应。

软件中断(SWI)异常发生在执行 SWI 指令,且没有更高优先级的异常标志置位的情况下。在进入相应处理程序时,cpsr 将被设置成管理模式。

如果系统使用嵌套 SWI 调用,则必须在跳转到嵌套的 SWI 之前,保存链接寄存器 r14 和 spsr 的值,以免其遭到破坏。

当一条不属于 ARM 或 Thumb 指令集的指令到达流水线的执行阶段时,若此时没有其它异常发生,就会产生未定义指令异常。ARM 处理器会“询问”协处理器,看它能否将其当作一条协处理器指令来处理。由于协处理器在流水线之后,所以指令确认可以在内核的执行阶段进行。如果这条指令不属于任何一个协处理器,则会产生未定义指令异常。

SWI 和未定义指令异常享有相同的优先级,因此不能同时发生。换句话说,正在执行的指令不可能既是一条 SWI 指令,又是一条未定义指令。

9.1.4 链接寄存器偏移

当一个异常发生时,链接寄存器就设置成基于当前 pc 值的一个特定地址。例如,当发生一个 IRQ 异常时,链接寄存器 lr 指向最后执行的指令地址加上 8。应确保异常处理程序不会破坏 lr,因为 lr 保存的是异常处理程序的返回地址。只有在当前指令执行完毕后,才

^{*} 这里的 FIQ 线是指 ARM 内核的引线。对于一个具体的 ARM 芯片,外部中断包括片上外设产生的中断。——译者注

ARM 嵌入式系统开发

进入 IRQ 异常处理,所以返回地址应指向下一条指令,即(lr-4)处。表 9.4 提供了一组对应不同异常的有用地址。*

表 9.4 基于链接寄存器的有用地址

异常	地址	用法
复位	—	复位没有定义 lr
数据中止	lr-8	指向导致数据中止异常的那条指令
FIQ	lr-4	FIQ 处理程序的返回地址
IRQ	lr-4	IRQ 处理程序的返回地址
预取指中止	lr-4	指向导致预取指中止异常的那条指令
SWI	lr	指向 SWI 指令的下一条指令
未定义指令	lr	指向未定义指令的下一条指令

接下来的 3 个例子显示了从 IRQ 或 FIQ 异常处理程序返回的不同方法。

【例 9.2】 说明使用 SUBS 指令从 IRQ 和 FIQ 处理程序返回的一种典型方法。

```
handler
    <handler code>
    ...
    SUBS    pc,r14,#4           ;pc = r14 - 4
```

因为在 SUB 指令尾部有一个 S,并且 pc 是目的寄存器,所以 cpsr 将自动从 spsr 寄存器中恢复。

【例 9.3】 说明另一种方法——在处理程序开头处从链接寄存器 lr 减去偏移量。

```
handler
    SUB     r14,r14,#4         ;r14 -= 4
    ...
    <handler code>
    ...
    MOVS    pc,r14             ;返回
```

服务完成后的返回是通过把链接寄存器 r14 的值写入 pc,同时从 spsr 寄存器中恢复 cpsr 来实现的。

* 这些是由 ARM 流水线的特性决定的。——译者注

【例 9.4】 使用中断堆栈来保存链接寄存器。

这种方法首先从链接寄存器减去一个偏移量,然后把它保存到中断堆栈里。

```
handler
    SUB    r14,r14,#4                ;r14 -= 4
    STMFD  r13!,{r0 - r3,r14}      ;保护上下文
    ...
    <handler code>
    ...
    LDMFD  r13!,{r0 - r3,pc}^       ;返回
```

为了返回到正常的执行,使用 LDM 指令来装载 pc。指令里的符号“^”强迫 cpsr 从 spsr 寄存器中恢复。

9.2 中 断

ARM 处理器有 2 种类型的中断。第一类是由外设引起的,即 IRQ 和 FIQ。第二类是一条引发中断的特殊指令——SWI 指令。2 种中断都会挂起正常的程序执行。

本节的重点将放在 IRQ 和 FIQ 中断上,包括下列要点:

- 分配中断;
- 中断延迟;
- IRQ 和 FIQ 异常;
- 基本的中断堆栈设计及实现。

9.2.1 分配中断

系统设计时可决定哪些硬件外设可以产生哪种中断请求。这种决定可通过硬件、软件(或两者)来实现,并依赖于所使用的嵌入式系统特性。

中断控制器用于连接多个外部中断到 ARM 两个中断请求之一。复杂的控制器可以通过编程来选择、决定一个外部中断源产生的是 IRQ 还是 FIQ 中断。

在分配中断时,系统设计者应采用一些标准的设计惯例。

- 软件中断通常被保留,用来调用特权操作系统例程。例如,可以使用 SWI 指令改变一个在用户模式下运行的程序到一个特权模式。关于 SWI 处理程序的例子,可参见第 11 章。

ARM 嵌入式系统开发

- 中断请求(IRQ)通常分配给通用中断。例如,一个周期性的定时器中断用来强制进行上下文切换,这往往就是一个 IRQ 异常。IRQ 异常的优先级比 FIQ 异常低,并且中断延迟也更长(这将在下一节中讨论)。

- 每个快速中断请求(FIQ)通常为要求快速响应的单个中断源保留。例如,直接存储器访问(DMA)专门用来传送存储器内的数据块,因此在一个嵌入式操作系统设计中,FIQ 异常一般用在专用场合,而 IRQ 异常则更多地用于操作系统的通用操作。

9.2.2 中断延迟

在中断驱动的嵌入式系统中,对中断延迟时间要求很苛刻。中断延迟是指:从外部中断请求信号发出到取出对应的中断服务程序(ISR)的第一条指令,这期间的间隔时间。

中断延迟依赖于软件与硬件的组合。体系结构设计师必须平衡系统设计,使多个并发的中断源能够得到处理,同时最小化中断延迟。如果中断不能得到及时处理,则系统会显现出很慢的响应,甚至导致系统瘫痪。

软件处理程序有 2 种主要的方法来缩短中断延迟。第一种方法是使用**嵌套中断**,它允许正在为一个中断服务时,再次响应其它中断(见图 9.3)。通过在某中断源得到服务后尽快重新允许中断,而不是等中断处理全部完成后再允许中断,可以实现这种方法。一旦嵌套的中断服务完成,控制权又回到前一个中断服务例程。

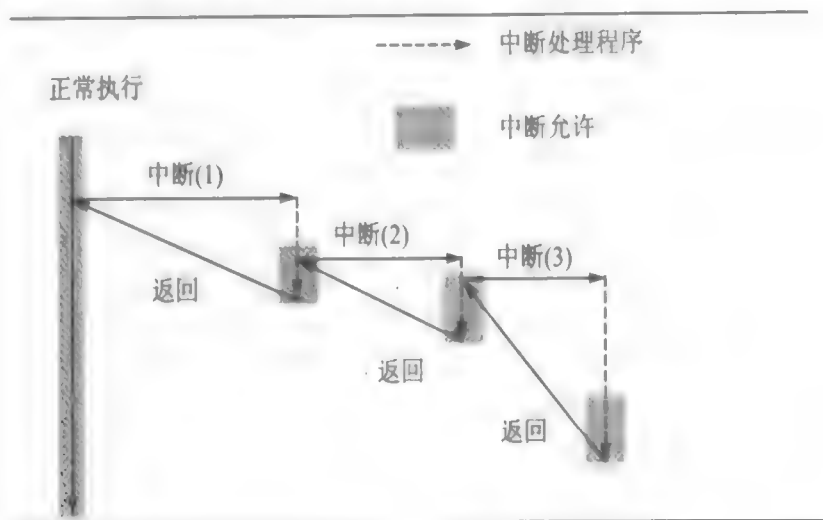


图 9.3 3 级嵌套中断

第二种方法是引入**优先级(prioritization)**。通过编程**中断控制器**,使其忽略与正在处理的中断同级或更低优先级的中断,因此只有高优先级的中断可以打断正在执行的中断服务。设定这种工作方式后,需要在每个中断处理程序中重新允许中断(进入每个中断处理程序时自动关中断),这样高级的中断处理程序才能执行,并将先前正在处理的较低级中断

挂起。

处理器在高优先级中断来到之前,处理低优先级的中断,因而与低优先级中断相比,高优先级中断的平均中断延迟较短。所以通过加速完成对时间敏感的中断,可以缩短延迟。

9.2.3 IRQ 与 FIQ 异常

只有当 cpsr 中相应的中断屏蔽被清除时,才可能发生 IRQ 与 FIQ 异常。ARM 处理器在处理中断前,继续执行已经处于流水线“执行”阶段的指令——这是一个在设计具有确定性中断处理时的重要因素,因为有些指令在执行阶段需要多个周期来完成。

一个 IRQ 或 FIQ 异常会使处理器硬件经过以下的一个标准流程(假设中断未被屏蔽):

- ① 处理器切换到一个特定的中断请求模式,表明产生了中断;
- ② 前一个模式的 cpsr 被保存到新的中断请求模式的 spsr;
- ③ pc 被保存到新的中断请求模式的 lr;
- ④ 关中断——在 cpsr 中禁止 IRQ,或 IRQ 与 FIQ 两者都被禁止,这会立即阻止相同类型的中断请求被响应;
- ⑤ 处理器跳转到向量表中一个特定的入口。

这个流程根据产生的中断类型不同而稍有变化。以下将对 2 种中断举例说明。例 9.5 说明了一个 IRQ 异常发生时的情况;而例 9.6 说明了一个 FIQ 异常发生时的情况。

【例 9.5】 图 9.4 说明了处理器在用户模式时响应了 IRQ 异常的情况。处理器从状态 1 开始。在本例中,cpsr 里的 IRQ 和 FIQ 异常位都被使能了。

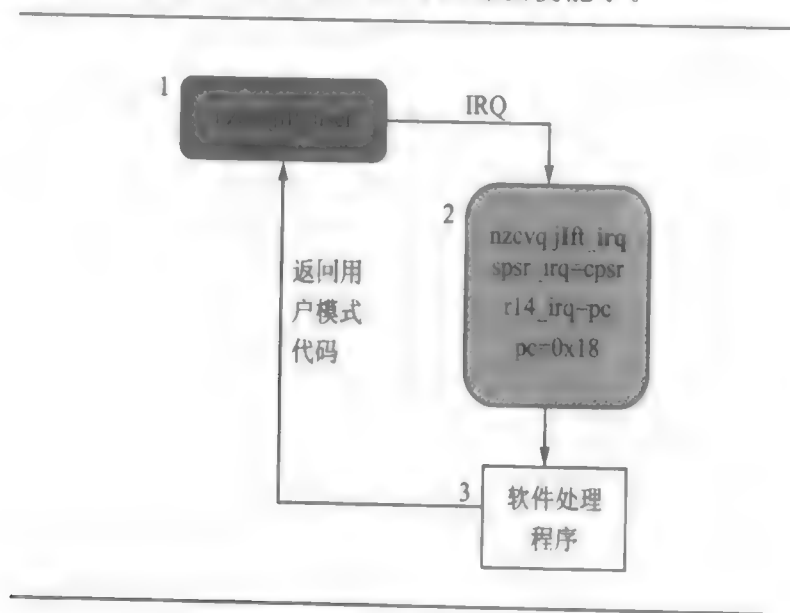


图 9.4 中断请求(IRQ)

ARM 嵌入式系统开发

当发生了 IRQ 后,处理器进入状态 2。这一转换自动将 IRQ 位置 1,关闭其它的 IRQ 异常。但是,FIQ 异常仍然是允许的,因为 FIQ 的优先级更高,当低优先级的 IRQ 异常发生时,不会被关闭。cpsr 里的处理器模式位变成 IRQ 模式。用户模式的 cpsr 自动复制到 spsr_irq。

发生中断时,寄存器 r14_irq 保存了 pc 的值。然后 pc 被置为向量表中 IRQ 的入口地址 0x18。

在状态 3 中,软件处理程序开始执行,并且调用适当的中断服务程序来为中断源服务。完成以后,处理器模式转换成状态 1 中最初的用户模式。

【例 9.6】 图 9.5 说明了一个 FIQ 异常的例子。

处理器也经历了与 IRQ 异常类似的过程,但与 IRQ 异常仅屏蔽其它的 IRQ 异常不同,处理器还屏蔽了其它的 FIQ 异常。这意味着进入状态 3 中的软件处理程序时,IRQ 和 FIQ 两种中断都被禁止了。

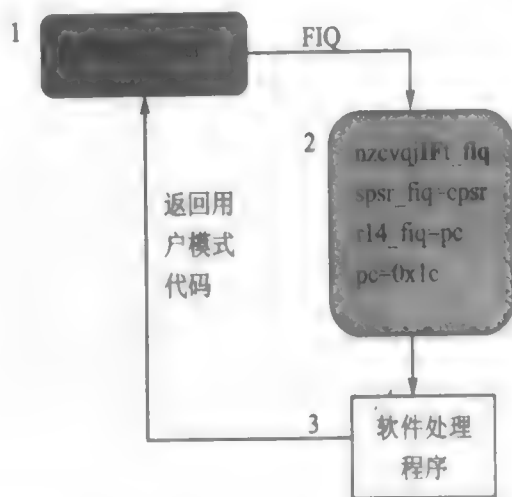


图 9.5 快速中断请求(FIQ)

变换到 FIQ 模式意味着没有必要保存寄存器 r8~r12,因为这些寄存器在 FIQ 模式下是自动备份保护的。这些寄存器可以用来保存诸如缓冲区指针或计数器之类的临时数据。这种特性使得 FIQ 对于处理单个中断源、高优先级、低延迟的中断是很理想的。

允许与禁止 FIQ 和 IRQ 异常

对 ARM 处理器内核有一种简单的方法,就是在处理器处于特权模式时,修改 cpsr,通过手工操作来允许和禁止中断。

表 9.5 说明了如何允许 IRQ 和 FIQ 中断。这一过程使用了 3 条 ARM 指令。

表 9.5 允许一个中断

cpsr 的值	IRQ	FIQ
之前	<i>nzcvqjIFt_SVC</i>	<i>nzcvqjIFt_SVC</i>
代码	enable_irq MRS r1, cpsr BIC r1, r1, #0x80 MSR cpsr_c, r1	enable_fiq MRS r1, cpsr BIC r1, r1, #0x40 MSR cpsr_c, r1
之后	<i>nzcvqjIfT_SVC</i>	<i>nzcvqjIfT_SVC</i>

第 1 条指令 MRS 把 cpsr 的内容复制到寄存器 r1;第 2 条指令清除 IRQ 或 FIQ 的屏蔽位;第 3 条指令把更新过的寄存器 r1 的内容复制回 cpsr,以允许中断请求。后缀 _c 表明被更新的位域是 cpsr 的控制域位[7:0](更多的细节参见第 2 章)。表 9.6 说明了禁止,即屏蔽中断请求的类似操作过程。

表 9.6 禁止一个中断

cpsr 的值	IRQ	FIQ
之前	<i>nzcvqjift_SVC</i>	<i>nzcvqjift_SVC</i>
代码	disable_irq MRS r1, cpsr ORR r1, r1, #0x80 MSR cpsr_c, r1	disable_fiq MRS r1, cpsr ORR r1, r1, #0x40 MSR cpsr_c, r1
之后	<i>nzcvqjIfT_SVC</i>	<i>nzcvqjIfT_SVC</i>

中断请求是允许还是禁止,只有在 MSR 指令已经完成了流水线的“执行”阶段后才确定,理解这一点是很重要的。在 MSR 指令完成执行前,中断仍可发生或者仍被屏蔽。

为了同时允许和禁止 IRQ 和 FIQ 异常,需要对第 2 条指令稍加改动。即把数据处理指令 BIC 或 ORR 后的立即数改为 0xc0,以同时允许或禁止这 2 种中断。

9.2.4 基本的中断堆栈设计与实现

异常处理程序广泛使用堆栈,每一种模式都有一个专门的寄存器保存堆栈指针。异常堆栈的设计取决于以下因素:

- 操作系统要求 每个操作系统对堆栈设计都有自己的要求;
- 目标硬件 目标硬件限制堆栈的实际空间大小和在存储器中的位置。

ARM 嵌入式系统开发

在堆栈设计时,须确定 2 点:

- **位置** 决定了在存储器映射中,堆栈从何处开始。大多数基于 ARM 系统设计的堆栈是采用向下递减式(下降式)的,栈顶位于存储器的高端地址。
- **堆栈大小** 依赖于处理程序的类型——嵌套的还是非嵌套的。一个嵌套中断处理程序需要更多的存储器空间,因为堆栈将随中断嵌套的深度而增加。

一个好的堆栈设计必须避免堆栈溢出——堆栈超出了分配给的存储空间,这会导致系统不稳定。有一些软件技术可以确认堆栈是否溢出,并在发生存储空间遭到无法恢复的破坏前,采取措施来修复堆栈。2 种主要的方法是:①使用存储保护;②在每个例程开始处调用堆栈检查函数。

IRQ 模式堆栈必须在中断允许前设置好——通常是在系统初始化代码中。在一个简单的嵌入式系统中,确认堆栈的大小是很重要的,因为堆栈大小在固件启动的初始阶段是被预留的。

图 9.6 显示了在线性空间中 2 种典型的存储安排方式。第一种方式 A,说明了一个传统的堆栈安排,中断的堆栈位于代码段之下。第二种方式 B,中断堆栈在用户堆栈之上,位于存储器的顶端。B 优于 A 之处是,B 在堆栈溢出时不会破坏向量表,因此系统在确认溢出后,还有机会纠正自己的错误。

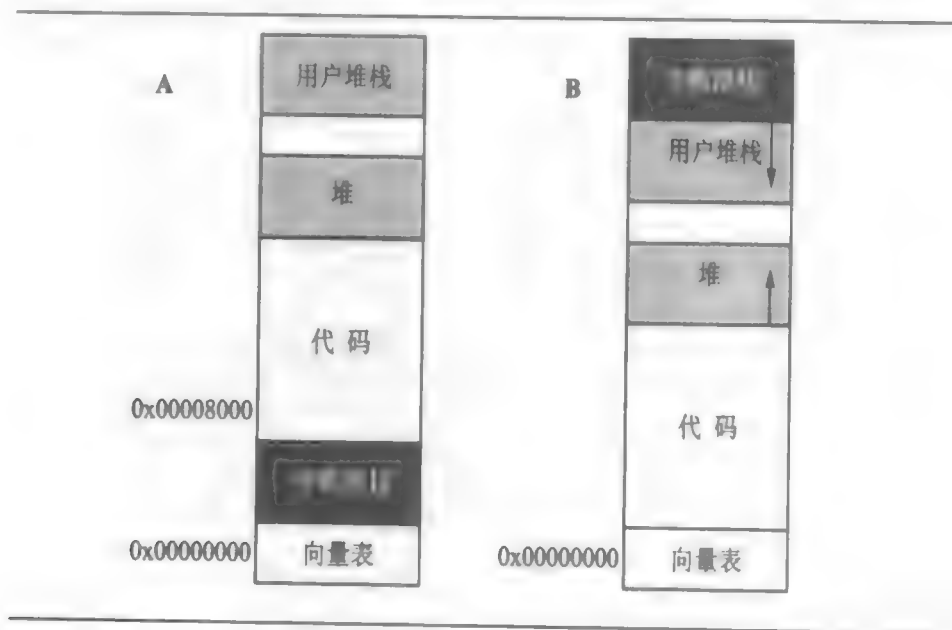


图 9.6 典型的存储器安排方式

【例 9.7】 每一种处理器模式都要建立一个堆栈,这是在处理器每次复位时完成的。图 9.7 说明了使用上述方式 A 的实现。为了有助于存储器安排,先声明一些存储区域名称的绝对地址映射定义。

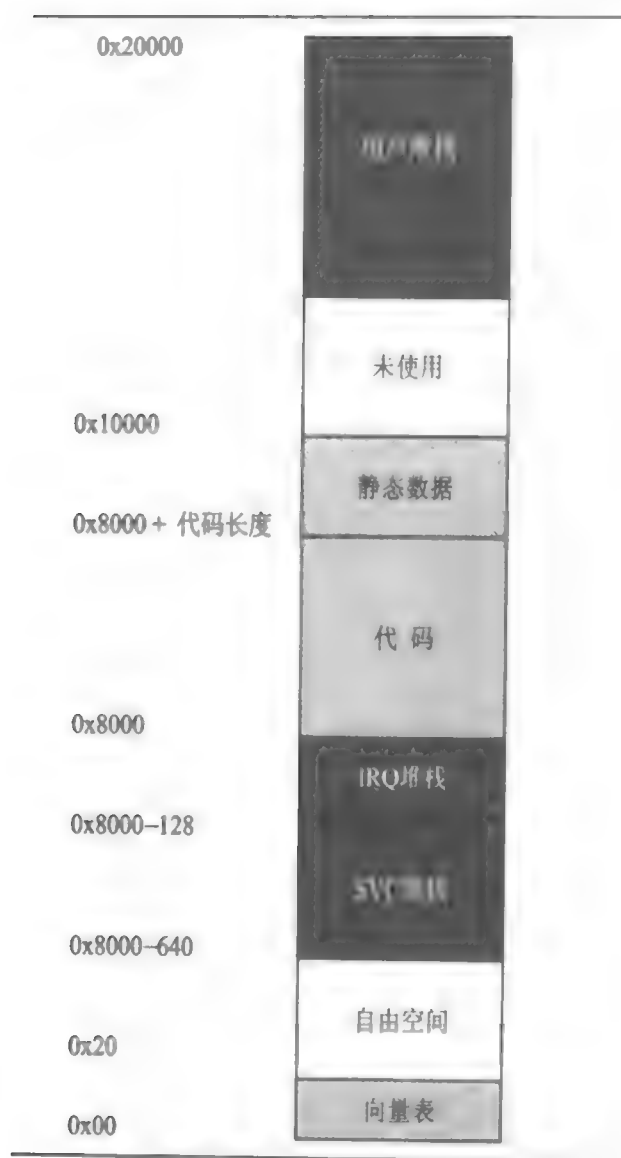


图 9.7 使用方式 A 的实现例子

例如,用户模式堆栈用标号 USR_Stack 表示,对应地址 0x20000;管理模式堆栈对应的地址是 IRQ 堆栈下面 128 字节处。

```

USR_Stack    EQU    0x20000
IRQ_Stack    EQU    0x8000
SVC_Stack    EQU    IRQ_Stack - 128
    
```

为了方便进行不同处理器模式的切换,另外声明了一组标号定义,使得每一种模式对应于一个特定的位格式。这些标号可以用来设置 cpsr 的新模式。

```

Usr32md     EQU    0x10    ;用户模式
FIQ32md     EQU    0x11    ;FIQ 模式
IRQ32md     EQU    0x12    ;IRQ 模式
    
```

ARM 嵌入式系统开发

SVC32md	EQU	0x13	;管理模式
Abt32md	EQU	0x17	;中止模式
Und32md	EQU	0x1b	;未定义指令模式
Sys32md	EQU	0x1f	;系统模式

出于安全,需要声明一个定义来禁止 cpsr 中的 IRQ 和 FIQ 异常:

NoInt	EQU	0xc0	;禁止中断
-------	-----	------	-------

NoInt 通过设置 2 个屏蔽位为 1 来禁止这 2 种中断。

初始化代码一开始就设置各个处理器模式的堆栈寄存器。当发生模式改变时,堆栈寄存器 r13 是受备份保护的寄存器之一。代码首先初始化 IRQ 堆栈。出于安全考虑,最好在 NoInt 与新模式之间,使用逻辑位“或”(OR)来确保中断是关闭的。

必须为每种模式设置堆栈。这里有一个例子,说明了处理器内核复位后是如何设置 3 种不同的堆栈的。

注意: 这是一个基本范例,没有实现中止、FIQ 及未定义指令异常模式的堆栈。如果需要,可以使用与之类似的代码来实现。

- **管理模式堆栈**——处理器内核从管理模式开始运行,因此 SVC 堆栈的建立包括把指向 SVC_NewStack 的地址装载到寄存器 r13_svc。对于本例,这个值为 SVC_Stack。

```

LDR    r13,SVC_NewStack    ;r13_svc
...
SVC_NewStack
DCD    SVC_Stack

```

- **IRQ 模式堆栈**——为了建立 IRQ 堆栈,处理器内核必须切换到 IRQ 模式。在寄存器 r2 存放 cpsr 的位格式,然后将其复制到 cpsr,就可以使处理器进入 IRQ 模式。这个操作使得寄存器 r13_irq 立即可用,并被赋值为 IRQ_Stack。

```

MOV    r2,#NoInt | IRQ32md
MSR    cpsr_c,r2
LDR    r13,IRQ_NewStack    ;r13_irq
...
IRQ_NewStack
DCD    IRQ_Stack

```

- **用户模式堆栈**——通常是最后设置的,因为当处理器处于用户模式时,没有直接修改 cpsr 的方法。由于系统模式与用户模式共享寄存器,所以可以强制处理器进入

系统模式来设置用户模式堆栈。

```
MOV    r2, # Sys32md
MSR    cpsr_c, r2
LDR    r13, USR_NewStack    ; r13_usr
...
USR_NewStack
DCD    USR_Stack
```

每种模式使用独立的堆栈而不是使用统一的堆栈处理,这样做的一个主要优点是:可以调试一个有错误的任务,并且与系统的其它部分隔离。

9.3 中断处理方法

最后这一节将介绍几种不同的中断处理方法*,包括从简单的非嵌套中断处理,到较复杂的分组优先级中断处理。每种方法都是以综述结合实例的形式给出的。

这里讨论的中断处理方法包括:

- **非嵌套中断处理** 顺序地处理和服务各个中断,这是最简单的中断处理程序;
- **嵌套中断处理** 处理多个没有分配优先级的中断;
- **可重入中断处理** 处理多个可以有优先级的中断;
- **优先级简单中断处理** 处理有优先级的中断;
- **优先级标准中断处理** 相对于低优先级中断,能在更短时间内处理高优先级中断;
- **优先级直接中断处理** 能在更短时间内处理高优先级中断,并直接跳转到一个专用的服务例程;
- **优先级分组中断处理** 一种中断处理方法,把中断划分到不同的优先级组中;
- **基于 VIC PL190 的中断服务例程** 说明了向量中断控制器(VIC)是如何改变中断服务例程设计的。

9.3.1 非嵌套中断处理

最简单的中断处理是非嵌套的:只有当控制权回到被中断的任务或过程时,才允许再次响应中断。由于一个非嵌套的中断处理程序在一个时段内只能为一个中断服务,所以这种

* 本节讨论的中断处理方法主要针对 ARM 内核中断。对于目前各厂家的许多 ARM 处理器芯片,一般都带有功能较强的(向量)中断控制器,可简化中断处理编程(参见文中最后一种中断处理方法)。——译者注

形式的中断处理程序不适合需要为多个不同优先级中断服务的复杂嵌入式系统。

图 9.8 显示了一个实现了简单的非嵌套中断处理系统上,发生一次中断的各种阶段:

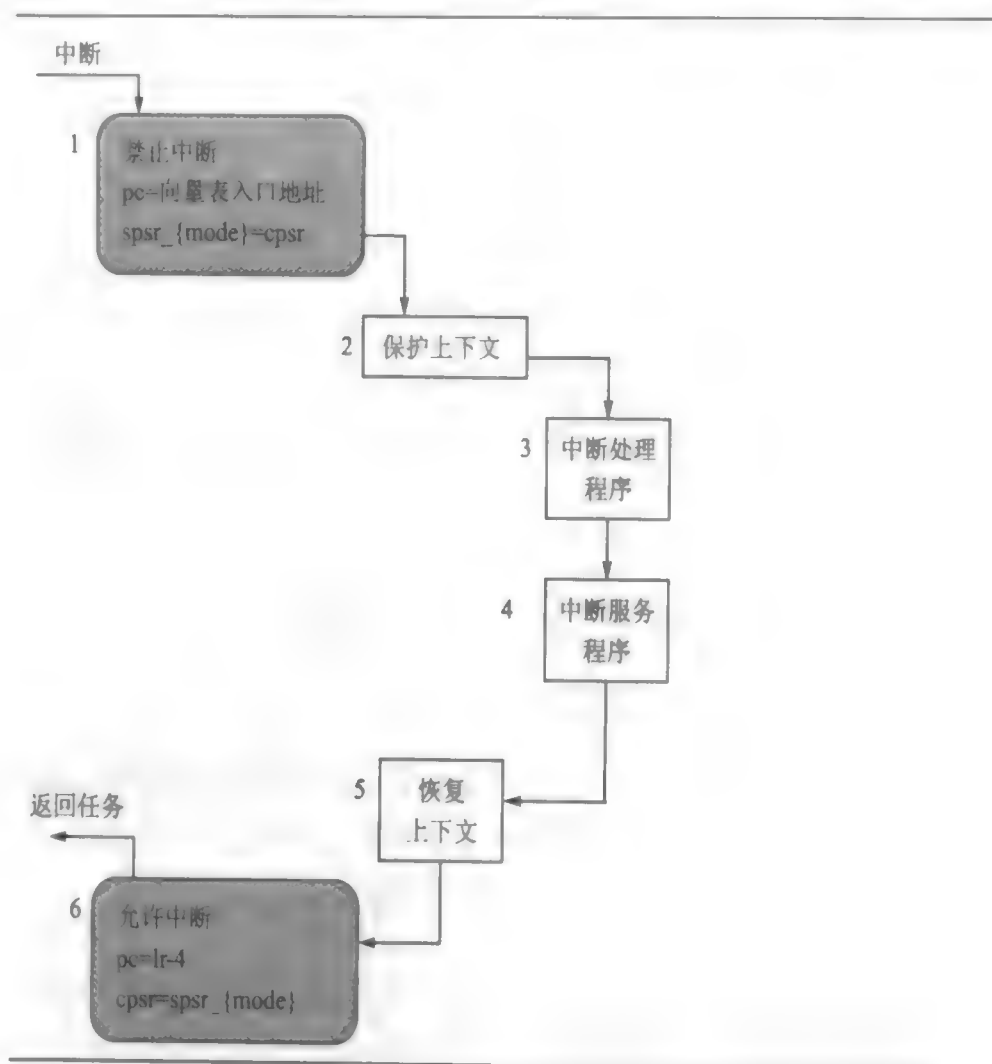


图 9.8 简单的非嵌套中断处理过程

① **禁止中断(关中断)**——当发生 IRQ 异常时,ARM 处理器将禁止其它 IRQ 异常的产生。处理器模式被设置为适当的中断请求模式,前一个模式的 cpsr 值被复制到新近有效的 spsr_{中断请求模式}。然后,处理器使 pc 指向向量表内正确的入口地址,并开始执行这一指令。这句指令将使 pc 指向专门的中断处理程序。

② **保存上下文**——进入处理程序,首先要保存当前模式下没有被自动分组保护的部分寄存器。

③ **中断处理程序**——处理程序确定外部中断源,并执行相应的中断服务例程(ISR)。

④ **中断服务程序**——ISR 为外部中断源提供服务,并复位该中断。

⑤ **恢复上下文**——从 ISR 返回到中断处理程序后,处理程序负责恢复上下文。

⑥ **允许中断(开中断)**——最后,为了从中断处理程序返回,使用 spsr_{中断请求模式}

的值恢复到 cpsr。接着 pc 指向响应中断时的下一条指令。

【例 9.8】 这个 IRQ 处理程序的例子假定初始化代码已经正确建立了 IRQ 堆栈。

```
interrupt_handler
    SUB      r14,r14,#4          ;调整 lr
    STMFD    r13!,{r0-r3,r12,r14} ;保护上下文
    <中断服务程序>
    LDMFD    r13!,{r0-r3,r12,pc}~ ;返回
```

第一条指令设置链接寄存器 r14_irq 的值为返回到被中断任务或过程的正确地址。如 9.1.4 小节描述的那样,由于流水线的特性,在每一个 IRQ 处理程序的入口,链接寄存器指向的地址比返回地址多 4 字节,因此处理程序必须从链接寄存器中减去 4。链接寄存器的值保存在堆栈里,为了返回被中断的任务,从堆栈恢复链接寄存器的内容,并将其复制到 pc。

注意:由于 ATPCS 调用规则,寄存器 r0~r3 及寄存器 r12 也被保存。这就允许一个遵循 ATPCS 的子程序在处理程序中被调用。

STMFD 指令通过把寄存器组的一个子集放置到堆栈,来保护上下文。由于执行一句 STMFD 或 LDMFD 指令的时间与参与的寄存器数目成正比,为了缩短中断延迟,只保存了尽可能少的寄存器。被保存到堆栈的寄存器由寄存器 r13_{中断请求模式}来指向。

如果在系统中使用的是高级语言,则理解编译器的过程调用规则是很重要的,因为这不仅会影响到被保存的寄存器,也会影响到其被保存到堆栈的顺序。例如,在一个子程序调用中,ARM 编译器保存了寄存器 r4~r11,因此没有必要再次保存,除非在中断处理程序中会用到它们。如果没有调用 C 例程,则没有必要保存全部的寄存器。只有当寄存器已被保存到中断堆栈后,调用一个 C 函数才是安全的。*

在一个非嵌套的中断处理程序中,没有必要保存 spsr,因为它不会被任何顺序的中断所破坏。

在中断处理程序的最后,指令 LDMFD 将恢复上下文,并从中断处理程序返回。LDMFD 指令末尾的“~”,意味着 cpsr 的值将从 spsr 中得到恢复,这只有在 pc 同时也被装载的情况下才有效。如果 pc 没有被装载,那么“~”只恢复用户模式的备份寄存器组。

在这里,中断处理程序完成了所有的事务,并且直接返回到应用程序。

一旦进入中断处理程序,保护上下文以后,处理程序就须确定中断源。以下给出一个简单的例子,以说明如何确定中断源。IRQStatus 是中断状态寄存器的地址。如果没有确定中断源,则控制权可以移交给另一个处理程序。在本例中,将控制权交给了 debug monitor。

* 因为编程者不能确定在一个 C 例程中究竟使用了哪些寄存器。——译者注

当然也可以忽略掉这个中断。

```

Interrupt_handler
    SUB        r14,r14,#4           ;r14 -= 4
    STMFED     sp!,{r0-r3,r12,r14} ;保护上下文
    LDR        r0,=IRQStatus        ;中断状态寄存器地址
    LDR        r0,[r0]              ;得到中断状态
    TST        r0,#0x0080           ;如果中断源是定时器
    BNE        timer_isr            ;就跳转到定时器 ISR
    TST        r0,#0x0001           ;如果按下按钮
    BNE        button_isr           ;则调用按钮 ISR
    LDMFED     sp!,{r0-r3,r12,r14} ;恢复上下文
    LDR        pc,=debug_monitor    ;否则跳转到 debug monitor

```

在代码中有两个 ISR: timer_isr 和 button_isr。它们与 IRQStatus 指向的中断状态寄存器中的特定位相对应,分别为 0x0080 和 0x0001。

小结 简单的非嵌套中断处理

- 依次处理各个中断;
- 中断延迟较大,不能在为一个中断服务的同时,处理新的中断;
- 优点:相对来说易于实现和调试;
- 缺点:不能用于处理有多个优先级中断的复杂嵌入式系统。

9.3.2 嵌套中断处理

嵌套中断处理考虑了在当前被调用的中断处理程序执行过程中,发生另一个中断的情形。在处理程序完成当前中断的服务前重新允许中断,可以实现中断嵌套。

对一个实时系统,这一特性增加了系统的复杂性,但也改进了系统的性能。复杂性的增加会引入一些非常细微的时序问题,这可能会导致系统失败,而且这些细微的问题非常难以解决。在设计一个嵌套中断方法时,应当非常仔细,以免出现这些问题。通过保护从中断中恢复的上下文,可以实现这一点,这样下一个中断就不会填满堆栈(导致堆栈溢出)或破坏任何寄存器的值。

任何一个中断处理程序的首要目标,是对中断的及时响应,所以中断处理程序不会等待任何异步的事件,也不会迫使其等待中断处理;第二个目标,是在为各中断提供服务时,不会延误常规同步代码的执行。

复杂性的增加,意味着设计者需要在效率和安全方面作出折衷。可以采用防御性的编码风格——假定会发生问题并采取相应措施。处理程序必须检查堆栈,并保护可能会被破

坏的寄存器。

图 9.9 显示了一个嵌套的中断处理过程。从图中可以看出,这个处理过程比 9.3.1 小节介绍的简单非嵌套中断处理过程要复杂得多。

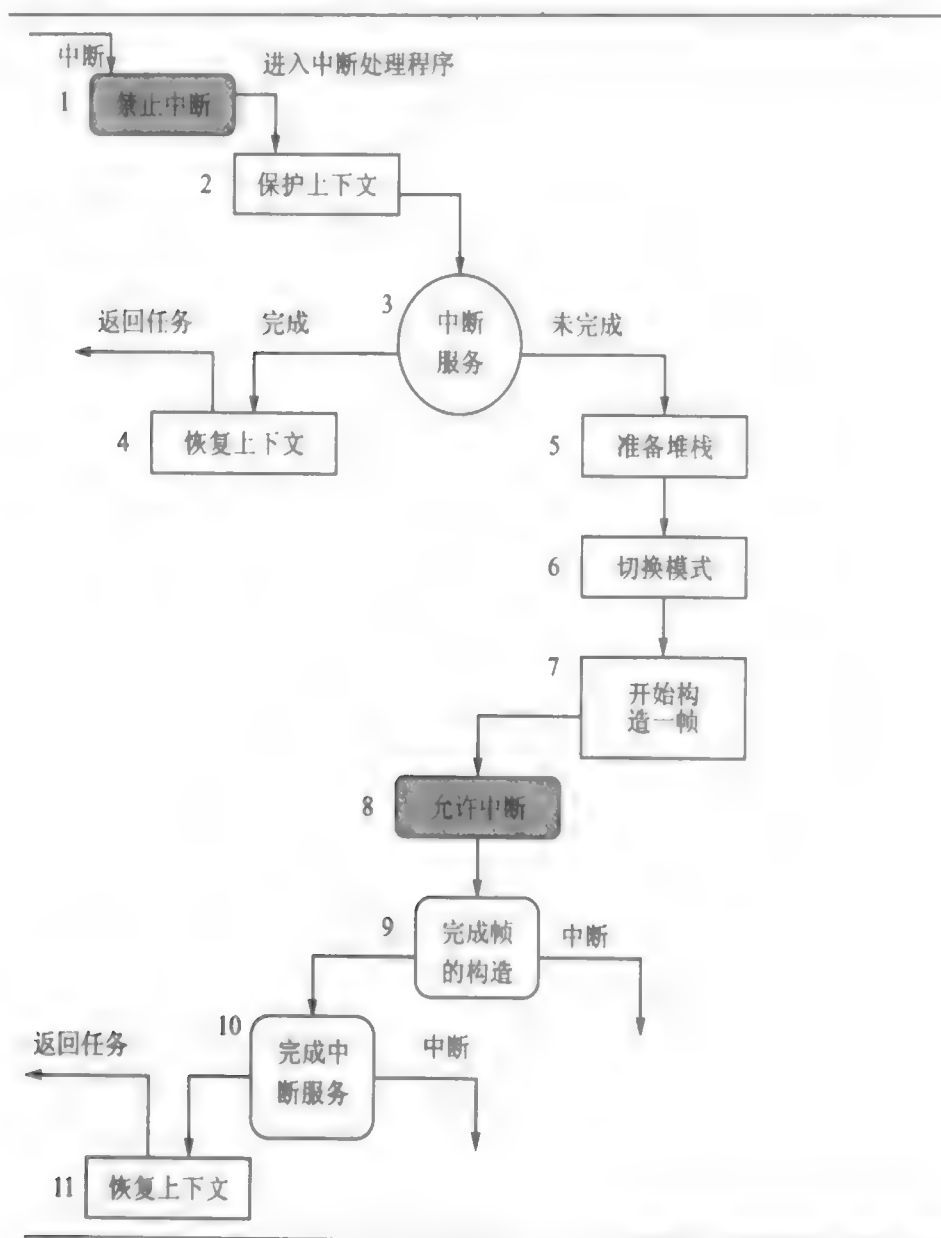


图 9.9 嵌套的中断处理过程

嵌套的中断处理程序入口处代码与简单的非嵌套中断处理程序类似,不同之处在于,在退出时,处理程序要测试被 ISR 更新过的一个标志。这个标志表明,是否需要做进一步的处理,如果不要求更多的处理,那么这个中断服务例程就完成了,处理程序也可以退出;如果需要进一步处理,处理程序可能要采取若干措施:重新允许中断,并/或执行一次上下文切换。

重新允许中断包括把 IRQ 模式切换到 SVC 或系统模式。在 IRQ 模式下,不能简单地

ARM 嵌入式系统开发

重新允许中断,因为这可能会导致链接寄存器 `r14_irq` 遭到破坏,特别是在执行完指令 `BL` 后即发生一个中断。这个问题将在 9.3.3 小节中详细讨论。

执行上下文切换包括复位(清空)IRQ 堆栈,因为当 IRQ 堆栈中还有数据时,处理程序不会执行上下文切换。所有保存在 IRQ 堆栈的寄存器必须转移到任务堆栈,典型地是放在管理模式堆栈上。然后,其余的寄存器也必须被保存到任务堆栈。在那里,它们会被转移到堆栈中一个称为堆栈帧的保留存储块上。

【例 9.9】 一个嵌套中断的处理程序范例,基于图 9.9 的流程图。

本小节的其余部分将详细介绍该处理程序及其各阶段的细节。

<code>Maskmd</code>	<code>EQU</code>	<code>0x1f</code>	;处理器模式屏蔽位
<code>SVC32md</code>	<code>EQU</code>	<code>0x13</code>	;SVC 模式
<code>I_Bit</code>	<code>EQU</code>	<code>0x80</code>	;IRQ 位
<code>FRAME_R0</code>	<code>EQU</code>	<code>0x00</code>	
<code>FRAME_R1</code>	<code>EQU</code>	<code>FRAME_R0 + 4</code>	
<code>FRAME_R2</code>	<code>EQU</code>	<code>FRAME_R1 + 4</code>	
<code>FRAME_R3</code>	<code>EQU</code>	<code>FRAME_R2 + 4</code>	
<code>FRAME_R4</code>	<code>EQU</code>	<code>FRAME_R3 + 4</code>	
<code>FRAME_R5</code>	<code>EQU</code>	<code>FRAME_R4 + 4</code>	
<code>FRAME_R6</code>	<code>EQU</code>	<code>FRAME_R5 + 4</code>	
<code>FRAME_R7</code>	<code>EQU</code>	<code>FRAME_R6 + 4</code>	
<code>FRAME_R8</code>	<code>EQU</code>	<code>FRAME_R7 + 4</code>	
<code>FRAME_R9</code>	<code>EQU</code>	<code>FRAME_R8 + 4</code>	
<code>FRAME_R10</code>	<code>EQU</code>	<code>FRAME_R9 + 4</code>	
<code>FRAME_R11</code>	<code>EQU</code>	<code>FRAME_R10 + 4</code>	
<code>FRAME_R12</code>	<code>EQU</code>	<code>FRAME_R11 + 4</code>	
<code>FRAME_PSR</code>	<code>EQU</code>	<code>FRAME_R12 + 4</code>	
<code>FRAME_LR</code>	<code>EQU</code>	<code>FRAME_PSR + 4</code>	
<code>FRAME_PC</code>	<code>EQU</code>	<code>FRAME_LR + 4</code>	
<code>FRAME_SIZE</code>	<code>EQU</code>	<code>FRAME_PC + 4</code>	
<code>IRQ_Entry</code>	;指令		状态 : 注释
<code>SUB</code>	<code>r14,r14,#4</code>		;2
<code>STMDB</code>	<code>r13!,{r0-r3,r12,r14}</code>		;2 : 保护上下文
<服务中断>			
<code>BL</code>	<code>read_RescheduleFlag</code>		;3 : 更多的处理
<code>CMP</code>	<code>r0,#0</code>		;3 : 是否要进一步处理?

```

LDMNEIA    r13!,{r0-r3,r12,pc}~      ;4   :否则返回
MRS        r2,spsr                    ;5   :复制 spsr_irq
MOV        r0,r13                      ;5   :复制 r13_irq
ADD        r13,r13,#6*4                ;5   :复位堆栈
MRS        r1,cpsr                    ;6   :复制 cpsr
BIC        r1,r1,#Maskmd               ;6
ORR        r1,r1,#SVC32md              ;6
MSR        cpsr_c,r1                   ;6   :切换到 SVC
SUB        r13,r13,#FRAME_SIZE-FRAME_R4 ;7   :保留空间
STMIA      r13,{r4-r11}                ;7   :保存 r4~r11
LDMIA      r0,{r4-r9}                  ;7   :恢复 r4~r9
BIC        r1,r1,#I_Bit                ;8
MSR        cpsr_c,r1                   ;8   :允许 IRA
STMDB      r13!,{r4-r7}                ;9   :保存 r4~r7 SVC
STR        r2,[r13,#FRAME_PSR]         ;9   :保存 PSR
STR        r8,[r13,#FRAME_R12]         ;9   :保存 r12
STR        r9,[r13,#FRAME_PC]          ;9   :保存 pc
STR        r14,[r13,#FRAME_LR]         ;9   :保存 lr
<完成中断服务程序>
LDMIA      r13!,{r0-r12,r14}           ;11  :恢复上下文
MSR        spsr_cxsf,r14               ;11  :恢复 spsr
LDMIA      r13!,{r14,pc}~              ;11  :返回

```

这个例子使用了一个堆栈帧结构。所有的寄存器,除了堆栈寄存器 r13 外,都被保存到这个帧里。这里,寄存器的顺序不是重要的;但 FRAME_LR 和 FRAME_PC 是例外,它们必须是这个帧里的最后 2 个寄存器,因为程序要使用单一的指令来返回:

```
LDMIA      r13!,{r14,pc}~
```

可能还有其它寄存器需要保存到堆栈帧里,这取决于使用的操作系统或应用程序。例如:

- 当操作系统支持 2 种模式——用户模式和管理(SVC)模式时,须保存寄存器 r13_usr 和 r14_usr;
- 当系统使用硬件浮点时,须保存浮点寄存器。

在本例中声明了一些定义,这些定义使不同的 cpsr/spsr 改变映射到一个特定的标号(例如 I_Bit)。

另外还声明了一组定义,定义了各个帧寄存器的帧指针偏移量。当重新允许中断,寄存

器必须被保存到堆栈帧中时,这些定义是很有用的。在本例中,堆栈帧是保存在 SVC 堆栈的。

本例的处理程序入口也使用了与简单非嵌套中断处理程序相同的代码。首先修改链接寄存器 r14,使其指向正确的返回地址;然后把上下文和链接寄存器 r14 保存到 IRQ 堆栈。

接下来中断服务程序就为中断提供服务。当服务完成或部分完成时,控制权又交还给处理程序。然后,处理程序调用一个称为 read_RescheduleFlag 的函数,该函数决定了是否需要做进一步的处理。如果不需要更多的处理,就在寄存器 r0 中返回一个非零的值;否则返回零。

注意:这里没有给出 read_RescheduleFlag 函数的源代码,因为它的实现要看具体情况而定。

接着测试寄存器 r0 的返回值,如果该寄存器不为 0,则处理程序恢复上下文,并把控制权交还给被挂起的任务。

如果寄存器 r0 被设置为 0,则表明需要做进一步的处理。第一步操作是保存 spsr,即把 spsr_irq 的副本送至寄存器 r2;然后,处理程序可以把 spsr 保存到堆栈帧。

由寄存器 r13_irq 指向的 IRQ 堆栈地址被复制到寄存器 r0,以备后用。下一步是复位(清空)IRQ 堆栈。这是通过在栈顶加 6×4 字节来实现的,因为堆栈是向下生长的,一条 ADD 指令可以用来复位堆栈。

处理程序无须担心保存在 IRQ 堆栈里的数据会被另一个嵌套的中断所破坏,因为这时中断还是被禁止的,处理程序一直要到 IRQ 堆栈的数据被恢复后,才重新允许中断。

然后,处理程序切换到 SVC 模式,中断仍被禁止。cpsr 被复制到寄存器 r1,并被修改,使处理器模式为 SVC。寄存器 r1 再被写回到 cpsr,于是当前模式就变成了 SVC 模式。新 cpsr 的副本保留在寄存器 r1 中,以便后面使用。

再下一步,是以堆栈帧的尺寸,通过扩展堆栈来构建一个堆栈帧。寄存器 r4~r11 可以保存在堆栈帧中,这样可以释放足够的寄存器,以恢复仍由寄存器 r0 指向的 IRQ 堆栈中的其余寄存器。

这一阶段堆栈帧保留的信息见表 9.7。仅有的、未被保存到堆栈帧中的寄存器,是在 IRQ 处理程序入口处已被保存的寄存器。

表 9.8 列出了对应于存在的 IRQ 寄存器, SVC 模式下的各寄存器。现在处理程序可以从 IRQ 堆栈中重新得到所有的数据,重新允许中断是安全的。

处理程序保存好重要的寄存器后,重新允许 IRQ 异常,并可以完成堆栈帧了。表 9.9 列出了一个完整的堆栈帧,它可以用在上下文切换或处理嵌套中断的场合。

表 9.7 SVC 堆栈帧

标 号	偏移量	寄存器	标 号	偏移量	寄存器
FRAME_R0	+0	—	FRAME_R8	+32	R8
FRAME_R1	+4	—	FRAME_R9	+36	r9
FRAME_R2	+8	—	FRAME_R10	+40	r10
FRAME_R3	+12	—	FRAME_R11	+44	r11
FRAME_R4	+16	r4	FRAME_R12	+48	—
FRAME_R5	+20	r5	FRAME_PSR	+52	—
FRAME_R6	+24	r6	FRAME_LR	+56	—
FRAME_R7	+28	r7	FRAME_PC	60	—

表 9.8 从 IRQ 堆栈取回数据

寄存器(SVC)	取回的 IRQ 寄存器
r4	r0
r5	r1
r6	r2
r7	r3
r8	r12
r9	r14(返回地址)

表 9.9 完整的堆栈帧

标 号	偏移量	寄存器	标 号	偏移量	寄存器
FRAME_R0	+0	r0	FRAME_R8	+32	r8
FRAME_R1	+4	r1	FRAME_R9	+36	r9
FRAME_R2	+8	r2	FRAME_R10	+40	r10
FRAME_R3	+12	r3	FRAME_R11	+44	r11
FRAME_R4	+16	r4	FRAME_R12	+48	r12
FRAME_R5	+20	r5	FRAME_PSR	+52	spsr_irq
FRAME_R6	+24	r6	FRAME_LR	+56	r14
FRAME_R7	+28	r7	FRAME_PC	+60	r14_irq

接下来这个阶段可以处理中断服务的其余部分。把当前任务控制块的寄存器 r13 的当前值保存,并从新的任务控制块装载一个新的值到寄存器 r13,就可以实现上下文切换。

最后,处理器可能返回到被中断的任务或处理程序;如果发生了上下文切换,则切换到另一个任务。

小 结 嵌套中断处理

- 处理没有分配优先级的多个中断;
- 中等或较高的中断延迟;
- 优点:在为一个中断服务完成前允许其它中断,以缩短中断延迟;
- 缺点:不处理中断的优先级,因此低优先级的中断会阻塞高优先级的中断。

9.3.3 可重入中断处理

可重入中断处理是处理多个中断的一种方法,这里,多个中断按优先级被排序、过滤筛选,这对于满足高优先级中断要求低延迟的需求是很重要的。这种类型的筛选是使用一般的嵌套中断处理无法做到的。

可重入中断处理程序与嵌套中断处理程序的基本区别是,在可重入中断处理程序中,更早地重新允许了中断,从而缩短了中断延迟。更早地重新允许中断会带来的一些问题,本节将作详细的讨论。

在可重入中断处理中的所有中断,必须是在 ARM 处理器的 SVC、系统、未定义指令或中止模式中被服务。

如果在一个中断模式中重新允许了中断,并且这个中断例程使用 BL 调用了一个子程序,那么这个子程序的返回地址将被设置在寄存器 r14_irq。这个地址随后会被另一个中断所破坏,因为寄存器 r14_irq 的值将被覆盖。为了避免这种情况,中断例程应切换到 SVC 或系统模式,那样,BL 指令可以使用寄存器 r14_svc 来保存子程序的返回地址。在通过 cpsr 重新允许中断前,中断源必须被禁止,这一般是通过设置中断控制器的某个寄存器位来实现的。

如果在中断源未被禁止,相应处理尚未完成前,重新允许中断,那么中断处理将会立即再次发生,导致无限的中断或紊乱情况(*race condition*)。大多数中断控制器都有一个中断屏蔽寄存器,用来屏蔽一个或多个中断,而其余的中断仍被允许。

由于中断是在 SVC 模式下得到服务的(例如,在任务堆栈),中断堆栈未被使用,所以 IRQ 堆栈寄存器 r13 可以用来指向一个 12 字节的结构体。这个结构体可以用于在中断入口处临时保存一些寄存器。

图 9.10 显示了可重入中断处理的流程图。

在一个可重入中断处理中,中断控制器对中断的优先级排序是非常重要的。如果中断

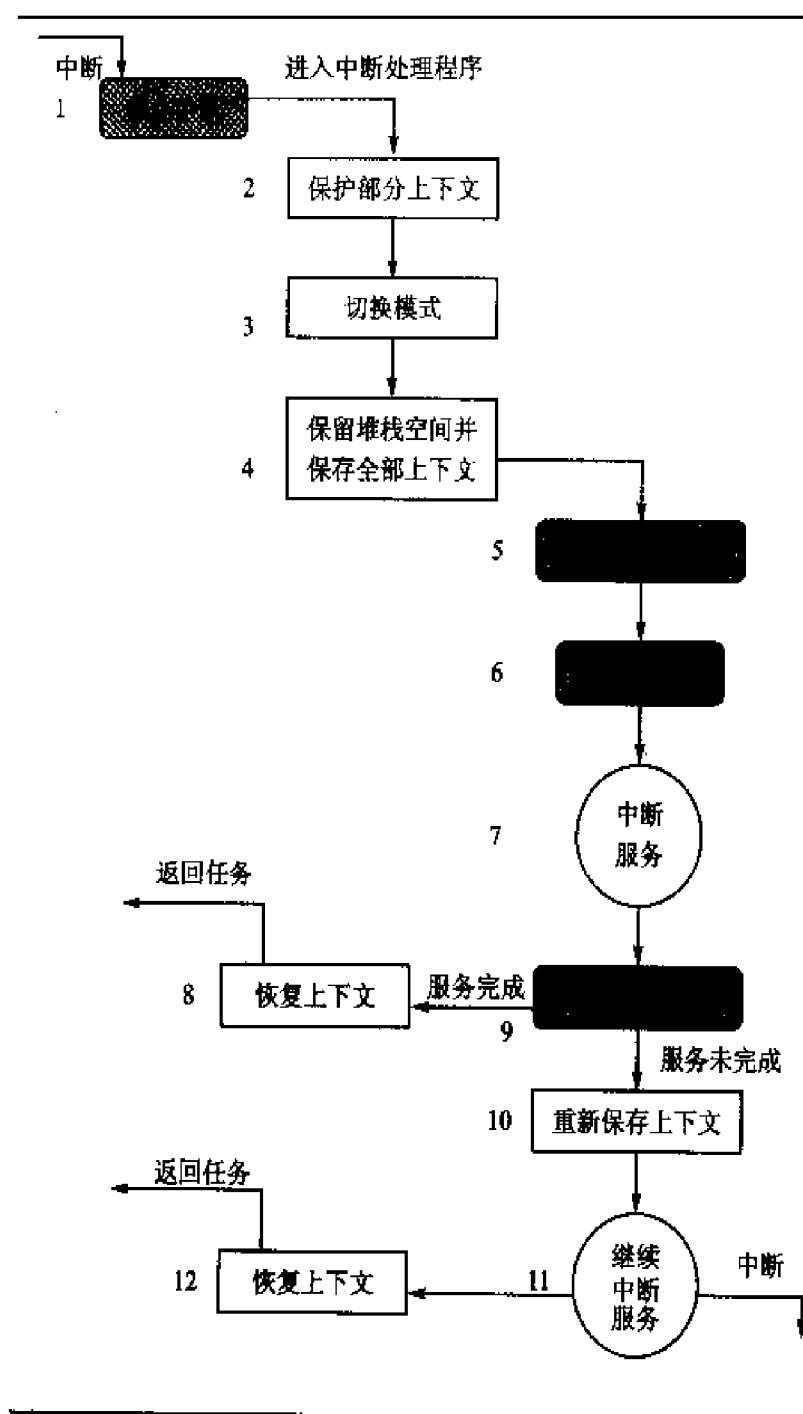


图 9.10 可重入中断处理

没有被优先级排序,则系统延迟性能会下降到与嵌套中断处理相同的水平。因为那样的话,低优先级中断能够抢占正在被服务的高优先级中断;同时,这也会导致在低优先级中断服务期间,高优先级中断得不到响应。

【例 9.10】 假设寄存器 `r13_irq` 已被设置为指向一个 12 字节的数据结构体,而不是指向标准的 IRQ 堆栈。`IRQ_spsr` 等偏移量用来指向这个结构的数据项。像所有的中断处理程序一样,需要做一些标准的定义,以修改寄存器 `cpsr` 和 `spsr`。

IRQ_R0	EQU	0	
IRQ_spsr	EQU	4	
IRQ_R14	EQU	8	
Maskmd	EQU	0x1f	;模式屏蔽位
SVC32nd	EQU	0x13	;SVC 模式
I_Bit	EQU	0x80	;IRQ 位
ic_base	EQU	0x80000000	
IRQStatus	EQU	0x0	
IRQRawStatus	EQU	0x4	
IRQEnable	EQU	0x8	
IRQEnableSet	EQU	0x8	
IRQEnableClear	EQU	0xc	
IRQ_Entry	; 指令	状态 ; 注释	
SUB	r14,r14,#4	;2 ;r14_irq = 4	
STR	r14,[r13,#IRQ_R14]	;2 ;保存 r14_irq	
MRS	r14,spsr	;2 ;复制 spsr	
STR	r14,[r13,#IRQ_spsr]	;2 ;保存 spsr	
STR	r0,[r13,#IRQ_R0]	;2 ;保存 r0	
MOV	r0,r13	;3 ;复制 r13_irq	
MRS	r14,cpsr	;3	
BIC	r14,r14,#Maskmd	;3	
ORR	r14,r14,#SVC32nd	;3	
MSR	cpsr_c,r14	;3 ;进入 SVC 模式	
STR	r14,[r13,# -8]!	;4 ;保存 r14	
LDR	r14,[r0,#IRQ_R14]	;4 ;r14_svc = r14_irq	
STR	r14,[r13,#4]	;4 ;保存 r14_irq	
LDR	r14,[r0,#IRQ_spsr]	;4 ;r14_svc = spsr_irq	
LDR	r14,[r0,#IRQ_R0]	;4 ;恢复 r0	
STMDB	r13!,{r0-r3,r8,r12,r14}	;4 ;保护上下文	
LDR	r14,=ic_Base	;5 ;中断控制器地址	
LDR	r8,[r14,#IRQStatus]	;5 ;装载中断状态	
STR	r8,[r14,#IRQEnableClear]	;5 ;清除中断	
MRS	r14,cpsr	;6 ;r14_svc = cpsr	
BIC	r14,r14,#I_Bit	;6 ;清除 I 位	

MSR	cpsr_c,r14	;6 :允许 IRQ 中断
BL	process_interrupt	;7 :调用 ISR
LDR	r14,=ic_base	;9 :中断控制器地址
STR	r8,[r14,#IRQEnableSet]	;9 :允许中断
BL	read_RescheduleFlag	;9 :更多的处理
CMP	r0,#0	;8 :是否进一步处理?
LDMNEIA	r13!,{r0-r3,r8,r12,r14}	;8 :装载上下文
MSRNE	spsr_cxsf,r14	;8 :更新 spsr
LDMNEIA	r13!,{r14,pc}~	;8 :返回
LDMIA	r13!,{r0-r3,r8}	;10 :否则装载寄存器
STMDB	r13!,{r0-r11}	;10 :保护上下文
BL	continue_servicing	;11 :继续服务
LDMIA	r13!,{r0-r12,r14}	;12 :恢复上下文
MSR	spsr_cxsf,r14	;12 :更新 spsr
LDMIA	r13!,{r14,pc}~	;12 :返回

处理程序的开头部分包括一个普通的中断入口,计算返回地址,为寄存器 r14_irq 的值减去 4。

现在重要的是,向由寄存器 r13_irq 指向的数据结构体的各个域赋值。被记录下来的寄存器有 r14_irq, spsr_irq 和 r0。在切换到 SVC 模式时,寄存器 r0 用来为数据结构体传递指针,因为寄存器 r0 是会被自动分组保护的。寄存器 r13_irq 不能用作此目的的原因是,它在 SVC 模式下是不可见的。

通过复制寄存器 r13_irq 的值到 r0 来保存指向数据结构体的指针。

偏移量(从 r13_irq 开始)	值
+0	r0(在入口)
+4	spsr_irq
+8	r14_irq

现在,处理程序将要使用操作 cpsr 的标准方法来使处理器进入 SVC 模式。SVC 模式的链接寄存器 r14 保存在 SVC 堆栈。堆栈指针减去 8,为 2 个 32 位字提供了堆栈空间。

然后,恢复寄存器 r14_irq,并保存到 SVC 堆栈。此刻 IRQ 和 SVC 的链接寄存器都保存在 SVC 堆栈里。

其余的 IRQ 上下文从数据结构体中被恢复出来,并传递到 SVC 模式。寄存器 r14_svc 现在保存的是 IRQ 模式下的 spsr。

接着,寄存器被保存到 SVC 堆栈。寄存器 r8 用来保存中断的屏蔽码,这些中断是被处理程序禁止的,它们将在后面的程序中被重新允许。

然后,中断源被清除。此时,系统(中断控制器)要对中断进行优先级排序,屏蔽所有低于当前优先级的中断,以免其打断高优先级的中断服务。中断优先级排序将在本章的稍后讨论。

中断源被清除后,重新允许 IRQ 异常就安全了。这是通过清除 cpsr 中的 I 位来实现的。

注意: 中断控制器仍然禁止外部中断。

现在可以处理(服务)这个中断了。中断处理过程不应该进行上下文切换,因为外部中断源是被禁止的。如果需要在中断处理过程中进行上下文切换,则应设置一个标志,以便中断处理程序能获知该情况。现在,重新允许外部中断是安全的。

处理程序须检查是否需要做进一步的处理。如果寄存器 r0 的返回值是非 0 的,则不需要再做处理;如果为 0,则处理程序恢复上下文并返回到被挂起的任务。

现在需要创建一个堆栈帧来完成服务例程。这是通过恢复部分上下文,然后把完整的上下文存储到 SVC 堆栈来实现的。

调用子程序 `continue_servicing`,完成对中断的服务。由于具体情况各异,本例没有提供该子程序的源代码。

完成中断例程服务后,控制权交还给被挂起的任务。

小 结 可重入中断处理

- 处理多个优先级被排序的中断;
- 较低的中断延迟;
- 优点:根据优先级处理中断;
- 缺点:变得更复杂。

9.3.4 优先级简单中断处理

非嵌套与嵌套中断处理都以先来先服务的原则为中断提供服务。与之相比,有优先级的中断处理程序,将一个特定的中断源与优先级联系起来,优先级用来指示中断得到服务的次序。因此,一个高优先级的中断将比一个低优先级的中断优先得到服务,这在许多嵌入式系统中是必需的。

进行优先级排序(prioritization)可以由硬件或软件来完成。对于硬件优先级处理,程序设计比较容易,因为中断控制器会提供当前需要服务的最高优先级的中断。这些系统在启动时要求更多的初始化代码,因为在系统正常运行前,必须构造中断和与之相联系的优先级表。而对于软件优先级处理,也需要外部中断控制器的协助,但这种中断控制器只提供一些最基本的功能,包括设置和清除屏蔽码,读中断状态和中断源。

本小节将介绍一种精选的软件优先级处理技术,因为它是一种通用的方法,不依赖某个特定的中断控制器。为了有助于描述这个优先级中断处理程序,而不涉及具体的处理器芯片,这里引入一个基于 ARM 标准中断控制器的虚构的(fictional)中断控制器。这个控制器连接多个中断源,并根据中断源是否被允许来产生一个 IRQ 或 FIQ 虚拟信号。

图 9.11 显示了一个简单优先级中断处理程序的流程图,它是基于可重入中断处理程序的。

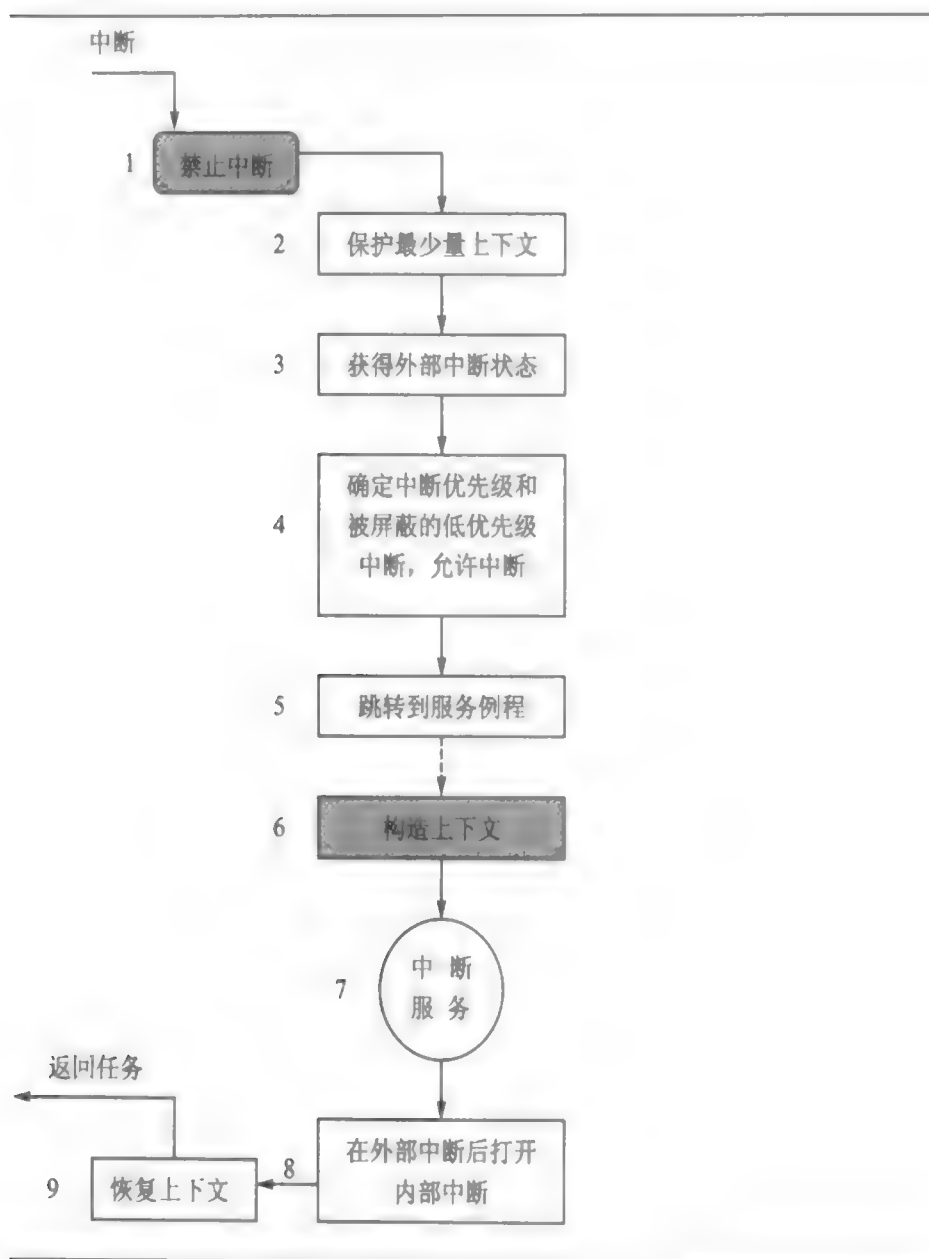


图 9.11 优先级中断处理程序

【例 9.11】 这个中断控制器有一个寄存器(IRQRawStatus)来保存原始的中断状态——被中断控制器屏蔽之前的中断信号状态。IRQEnable 寄存器确定哪个中断是被处理

ARM 嵌入式系统开发

器屏蔽的,这个寄存器只能用 IRQEnableSet 和 IRQEnableClear 来设置或清除。表 9.10 说明了中断控制器中的各个寄存器名称、从控制器基地址开始的偏移量、读/写操作和寄存器的描述。

表 9.10 中断控制器寄存器

寄存器	偏移量	读/写	描 述
IRQRawStatus	+0x04	r	表示中断源的状态
IRQEnable	+0x08	r	屏蔽向内核产生 IRQ 或 FIQ 的中断源
IRQStatus	+0x00	r	表示屏蔽后的中断源状态
IRQEnableSet	+0x08	w	中断使能寄存器置位
IRQEnableClear	+0x0c	w	中断使能寄存器清零

```
I_Bit      EQU 0x80

PRIORITY_0  EQU 2                ;Comms Rx
PRIORITY_1  EQU 1                ;Comms Tx
PRIORITY_2  EQU 0                ;Timer 1
PRIORITY_3  EQU 3                ;Timer 2

BINARY_0    EQU 1 << PRIORITY_0 ;1 << 2 0x00000004
BINARY_1    EQU 1 << PRIORITY_1 ;1 << 1 0x00000002
BINARY_2    EQU 1 << PRIORITY_2 ;1 << 0 0x00000001
BINARY_3    EQU 1 << PRIORITY_3 ;1 << 3 0x00000008

MASK_3      EQU BINARY_3
MASK_2      EQU MASK_3 + BINARY_2
MASK_1      EQU MASK_2 + BINARY_1
MASK_0      EQU MASK_1 + BINARY_0

ic_Base     EQU 0x80000000
IRQStatus   EQU 0x0
IRQRawStatus EQU 0x4
IRQEnable    EQU 0x8
IRQEnableSet EQU 0x8
IRQEnableClear EQU 0xc
```

IRQ_Handler	指令	状态 ; 注释
	SUB r14,r14,#4	;2 : r14_irq -= 4
	STMED r13!, {r14}	;2 : 保存 r14_irq
	MRS r14,spsr	;2 : 复制 spsr_irq
	STMED r13!, {r10,r11,r12,r14}	;2 : 保护上下文
	LDR r14, = ic_Base	;3 : 中断控制器地址
	MOV r11, #PRIORITY_3	;3 : 默认优先级
	LDR r10, [r14, #IRQStatus]	;3 : 装载 IRQ 状态
	TST r10, #BINARY_3	;4 : 如果中断源为定时器 2
	MOVNE r11, #PRIORITY_3	;4 : 则优先级为 P3(lo)
	TST r10, #BINARY_2	;4 : 如果中断源为定时器 1
	MOVNE r11, #PRIORITY_2	;4 : 则优先级为 P2
	TST r10, #BINARY_1	;4 : 如果中断源为通信发送
	MOVNE r11, #PRIORITY_1	;4 : 则优先级为 P1
	TST r10, #BINARY_0	;4 : 如果中断源为通信接收
	MOVNE r11, #PRIORITY_0	;4 : 则优先级为 P0(hi)
	LDR r12, [r14, #IRQEnable]	;4 : IRQEnable 寄存器
	ADR r10, priority_masks	;4 : 屏蔽码地址
	LDR r10, [r10, r11, LSL #2]	;4 : 优先级
	AND r12, r12, r10	;4 : AND 允许寄存器
	STR r12, [r14, #IRQEnableClear]	;4 : 清除中断
	MRS r14, cpsr	;4 : 保存 CPSR
	BIC r14, r14, #I_Bit	;4 : 清零 I 位
	MSR cpsr_c, r14	;4 : 允许 IRQ 中断
	LDR pc, [pc, r11, LSL #2]	;5 : 跳转到一个 ISR
	NOP	
	DCD service_timer1	;定时器 1 ISR
	DCD service_commtx	;通信发送 ISR
	DCD service_commr	;通信接收 ISR
	DCD service_timer2	;定时器 2 ISR

priority_masks

DCD	MASK_2	;优先级屏蔽码 2
DCD	MASK_1	;优先级屏蔽码 1
DCD	MASK_0	;优先级屏蔽码 0
DCD	MASK_3	;优先级屏蔽码 3

...

```
service_timer1
    STMFD    r13!,{r0-r9}           ;6: 保护上下文
    ;<服务例程>
    LDMFD    r13!,{r0-r10}          ;7: 恢复上下文
    MRS      r11,cpsr                ;8: 复制 cpsr
    ORR      r11,r11, #I_Bit         ;8: 设置 I 位
    MSR      cpsr_c,r11              ;8: 禁止 IRQ
    LDR      r11, =ic_Base           ;8: 中断控制器地址
    STR      r12,[r11,#IRQEnableSet] ;8: 允许中断
    LDMFD    r13!,{r11,r12,r14}      ;9: 恢复上下文
    MSR      spsr_cxsf,r14           ;9: 设置 spsr
    LDMFD    r13!,{pc}~              ;9: 返回
```

对于 FIQ 异常,大多数中断控制器还有一组相应的寄存器,甚至允许单独的中断源使用特定的信号来申请内核中断。因此,通过编程中断控制器,一个特定的中断源就可以产生一个 IRQ 或 FIQ 异常。

在存储空间中,寄存器位于从某个基地址开始的特定偏移量处。表 9.10 列出了以中断控制器基地址 ic_Base 开始的各寄存器偏移量。

注意: IRQEnable 与 IRQEnableSet 都使用偏移量 0x08。

在中断控制器中,每一位都与一个特定的中断源关联起来(见图 9.12)。例如,位 2 对应的是串口通信的接收中断。

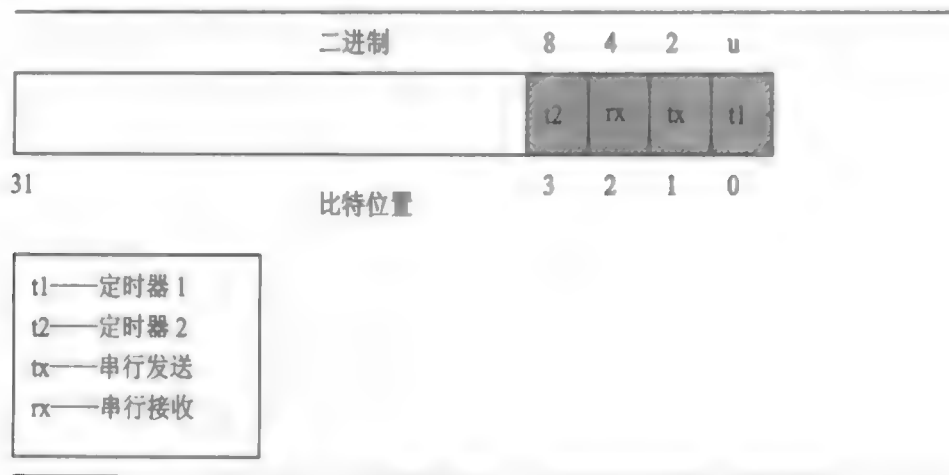


图 9.12 32 位中断控制寄存器

PRIORITY_x 定义了在本例中用到的 4 个中断源,并使其分别对应于不同的中断优先级。PRIORITY_0 是优先级最高的中断,PRIORITY_3 是优先级最低的中断。

BINARY_x 为每种优先级提供了位格式。例如,对于一个 PRIORITY_0 的中断,其

二进制位格式是 $0x00000004$ (即 $1 \ll 2$)。对每个优先级来说,都有一个相应的屏蔽码,用来屏蔽优先级相同或更低的中断。例如,MSK_2 会屏蔽定时器 2 (优先级为 3) 和串口接收 (优先级为 2) 中断。

中断控制器的地址定义也列出来了。ic_Base 是基地址,其余的定义 (如 IRQStatus) 都是以这个基地址开始的偏移量。

优先级中断处理程序从标准的入口开始,但起初只有 IRQ 链接寄存器被保存到 IRQ 堆栈里。

接下来处理程序得到 spsr,并将其内容保存在寄存器 r14_irq,释放一组寄存器,以便在进行优先级排序 (prioritization) 时使用。

处理程序需要获得中断控制器的状态,这是通过装载中断控制器的基地址到寄存器 r14,并用 ic_Base (寄存器 r14) 加偏移量 IRQStatus (0x00) 把状态信息装载到寄存器 r10 来实现的。

然后,处理程序需要通过测试状态信息来确定优先级最高的中断。如果一个特定的中断源与某个优先级相匹配,那么在寄存器 r11 中就设置这一优先级。这种方法把中断源与所有设置的优先级进行比较,从最低优先级开始,一直到最高优先级。

经过这一段代码后,寄存器 r14_irq 将保存中断控制器的基地址,寄存器 r11 将保存优先级最高的中断的位号 (bit number)。现在重要的是禁止低优先级和同级的中断,而仍然允许更高优先级的中断打断这一处理程序。

注意: 这种处理方法具有更好的时间确定性,因为花在查找最高优先级中断的时间总是一样的。

为了设置控制器中的中断屏蔽码,处理程序必须确定当前 IRQ 使能寄存器的内容,并且获得优先级屏蔽码表的起始地址。priority_masks 定义在处理程序的最后。

寄存器 r12 现在保存的是 IRQ 使能寄存器的值,寄存器 r10 保存的是优先级表的起始地址。为了得到正确的屏蔽码,寄存器 r11 左移 2 位 (使用桶形移位器 LSL #2)。这使得地址乘以 4 并加到优先级表的起始地址上去。

寄存器 r10 包含了新的屏蔽码。下一步是使用这个屏蔽码来清除低优先级中断: 先把这一屏蔽码和寄存器 r12 (IRQEnable 寄存器的内容) 进行二进制逻辑“与”,再把结果写到寄存器 IRQEnableClear。现在通过清除 cpsr 中的 i 位来使能 IRQ 异常就安全了。

最后,处理程序跳转到正常的服务例程,这是通过修改寄存器 r11 (它仍然保存着最高优先级的中断) 和 pc 来实现的。直接把服务例程的地址装载到 pc (寄存器 r11 左移 2 位,即乘以 4,再与 pc 相加),就使处理程序跳转到正确的例程。

跳转表应紧挨着装载 pc 的指令。在操作 pc 的指令与跳转表之间有一条 NOP 指令,因为 pc 指向超前了 2 条指令 (即 8 字节)。优先级屏蔽码表是按中断源对应的位排序的。

每个 ISR 使用相同的入口处理方法,例子给出的是 timer1 中断服务程序。

在上述中断入口处理(头部)之后,是具体的 ISR。一旦 ISR 完成,中断源必须被复位(清除),并把控制权交还给被中断的任务。

在中断可以重新打开前,处理程序必须禁止 IRQ。现在,外部中断可以恢复成原值,这一点是可以做到的,因为服务例程没有修改寄存器 r12,它仍然保持最初的值。

为了返回到被中断的任务,需要恢复上下文,并把最初的 spsr 复制到 spsr_irq。

小 结 优先级简单中断处理

- 处理有优先级的中断;
- 较低的中断延迟;
- 优点:确定的中断延迟时间,首先要确定优先级,然后屏蔽较低优先级的中断,再调用服务;
- 缺点:获得一个低优先级的服务例程所用的时间,与一个高优先级的例程一样长。

9.3.5 优先级标准中断处理

从优先级简单中断处理继续发展下去,处理程序的复杂度进一步提高。优先级简单处理程序采用测试所有的中断来确定最高优先级——这是一种效率较低的方法;但也有其优点,即响应时间的确定性,因为每个中断优先级都会花费同样长的时间得到确认。

一种较好的替换办法就是,在高优先级中断被确认后,程序尽早地跳转(见图 9.13),一旦确认优先级,就设置 pc 立即跳转。这意味着对于优先级标准中断处理程序来说,代码对中断优先级的确认部分更为复杂。确认部分将决定优先级,并立即跳转到一个屏蔽低优先级中断的例程,然后再通过一个跳转表跳转到合适的 ISR。

【例 9.12】 一个优先级标准中断处理程序的开头部分与一个优先级简单中断处理程序类似,只是更早地拦截了高优先级的中断。寄存器 r14 用来指向中断控制器的基地址,寄存器 r10 用来保存中断控制器的状态寄存器内容。为了允许处理程序重定位,pc 指向的当前地址被保存到寄存器 r11。

现在可以通过比较最高到最低优先级来测试中断源了。第一个与中断源匹配的优先级决定了引入中断的优先级,因为每一个中断都预先设定了优先级。一旦匹配成功,处理程序就可以跳转到屏蔽低优先级中断的程序中。

I_Bit	EQU	0x80	
PRIORITY_0	EQU	2	;通信接收 Rx
PRIORITY_1	EQU	1	;通信发送 Tx
PRIORITY_2	EQU	0	;定时器 1

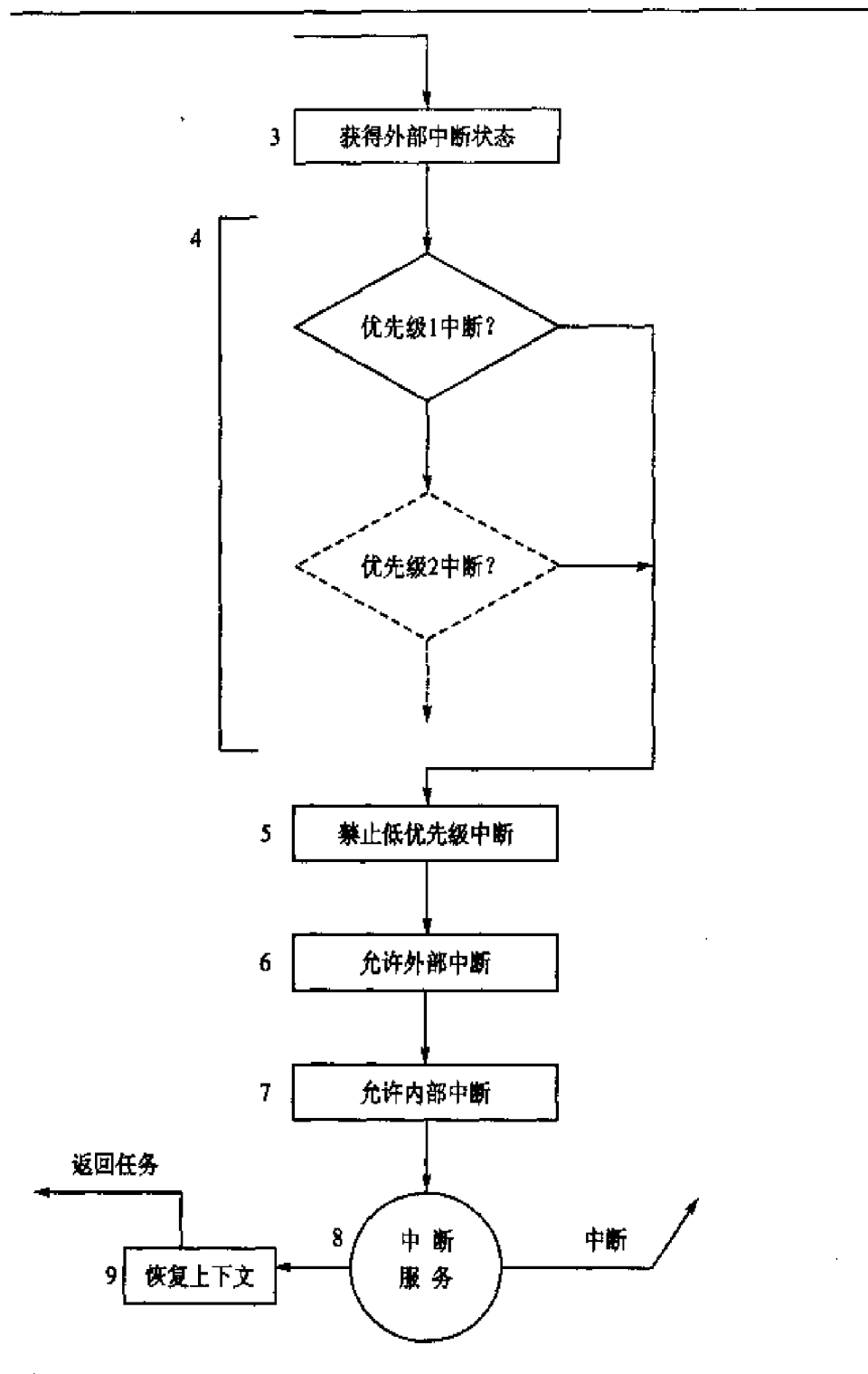


图 9.13 优先级标准中断处理的部分程序

PRIORITY_3	EQU	3	;定时器 2
BINARY_0	EQU	1 << PRIORITY_0	;1 << 2 0x00000004
BINARY_1	EQU	1 << PRIORITY_1	;1 << 1 0x00000002
BINARY_2	EQU	1 << PRIORITY_2	;1 << 0 0x00000001
BINARY_3	EQU	1 << PRIORITY_3	;1 << 3 0x00000008
MASK_3	EQU	BINARY_3	
MASK_2	EQU	MASK_3 + BINARY_2	
MASK_1	EQU	MASK_2 + BINARY_1	
MASK_0	EQU	MASK_1 + BINARY_0	
ic_Base	EQU	0x80000000	
IRQStatus	EQU	0x0	
IRQRawStatus	EQU	0x4	
IRQEnable	EQU	0x8	
IRQEnableSet	EQU	0x8	
IRQEnableClear	EQU	0xc	
IRQ_Handler	;指令		状态 :注释
SUB	r14,r14, #4		;2 : r14_irq -= 4
STMFD	r13!, {r14}		;2 : 保存 r14_irq
MRS	r14,spsr		;2 : 复制 spsr_irq
STMFD	r13!,{r10,r11,r12,r14}		;2 : 保护上下文
LDR	r14, = ic_Base		;3 : 中断控制器地址
LDR	r10,[r14, # IRQStatus]		;3 : 装载 IRQ 状态
MOV	r11,pc		;4 : 复制 pc
TST	r10, # BINARY_0		;5 : 如果中断源为通信接收
BLNE	disable_lower		;5 : 则跳转
TST	r10, # BINARY_1		;5 : 如果中断源为通信发送
BLNE	disable_lower		;5 : 则跳转
TST	r10, # BINARY_2		;5 : 如果中断源为定时器 1
BLNE	disable_lower		;5 : 则跳转
TST	r10, # BINARY_3		;5 : 如果中断源为定时器 2
BLNE	disable_lower		;5 : 则跳转
disable_lower			
SUB	r11,r14,r11		;5 : r11 = r14 - pc 的副本

```

LDR      r12, = priority_table      ;5 : 优先级表地址
LDRB     r11,[r12,r11,LSR #3]       ;5 : mem8[tbl + (r11 >> 3)]
ADR      r10,priority_masks         ;5 : 优先级屏蔽码地址
LDR      r10,[r10,r11,LSL #2]       ;5 : 装载优先级屏蔽码
LDR      r14, = ic_Base              ;6 : 中断控制器地址
LDR      r12,[r14, # IRQEnable]     ;6 : IRQ 允许寄存器
AND      r12,r12, r10               ;6 : AND 允许寄存器
STR      r12,[r14, # IRQEnableClear] ;6 : 允许新中断
MRS      r14,cpsr                   ;7 : 复制 cpsr
BIC      r14,r14, # I_Bit           ;7 : 清零 I 位
MSR      cpsr_c,r14                 ;7 : 允许 IRQ
LDR      pc,[pc,r11,LSL #2]         ;8 : 跳转到一个 ISR
NOP
DCD      service_timer1             ;定时器 1 ISR
DCD      service_commtx             ;通信发送 ISR
DCD      service_commr             ;通信接收 ISR
DCD      service_timer2             ;定时器 2 ISR

priority_masks
DCD      MASK_2                     ;优先级屏蔽码 2
DCD      MASK_1                     ;优先级屏蔽码 1
DCD      MASK_0                     ;优先级屏蔽码 0
DCD      MASK_3                     ;优先级屏蔽码 3

priority_table
DCB      PRIORITY_0                 ;优先级 0
DCB      PRIORITY_1                 ;优先级 1
DCB      PRIORITY_2                 ;优先级 2
DCB      PRIORITY_3                 ;优先级 3
ALIGN

```

为了禁止同级或低级优先级中断,处理程序进入一个例程,它首先使用寄存器 r11 中的基地址和链接寄存器 r14 来计算优先级。

完成 SUB 指令后,寄存器 r11 的值将为 4,12,20 或 28。这些值对应于中断优先级乘以 8 再加上 4。然后寄存器 r11 除以 8,并与 priority_table 的地址相加。LDRB 指令执行后,寄存器 r11 将为中断优先级号 0,1,2 或 3 之一。

现在可以确定中断屏蔽码了,r11 左移 2 位再加入到寄存器 r10(寄存器 r10 中保存了 pri-

ority_mask 的地址)。

中断控制器的基地址被复制到寄存器 r14_irq, 用来读取控制器中 IRQEnable 寄存器的内容, 并存入寄存器 r12。

寄存器 r10 包含了新的屏蔽码。下一步是使用这个屏蔽码来清除低优先级中断: 先把这一屏蔽码和寄存器 r12 (IRQEnable 寄存器的内容) 进行二进制逻辑“与”, 再把结果写到寄存器 IRQEnableClear。现在通过清除 cpsr 中的 i 位来使能 IRQ 异常就安全了。

最后, 处理程序跳转到正常的服务例程, 这是通过修改寄存器 r11 (它仍然保存着最高优先级的中断) 和 pc 来实现的。直接把服务例程的地址装载到 pc (寄存器 r11 左移 2 位, 即乘以 4, 再与 pc 相加), 就使处理程序跳转到正确的例程。跳转表必须跟在装载 pc 的指令后面。在跳转表与 LDR 指令间有一条 NOP 指令, 因为 pc 的指向超前 2 条指令 (即 8 字节)。

注意: 优先级屏蔽码表是按中断位排序的, 而优先级表是按优先级次序排序的。

小结 优先级标准中断处理

- 进入高优先级的中断服务时间比进入低优先级的中断服务时间更短;
- 较低的中断延迟;
- 优点: 高优先级中断得到优先处理, 并不要复制代码来设置外部中断屏蔽;
- 缺点: 时间上有些损失, 因为这种处理程序需要 2 次跳转, 导致流水线在每次跳转发生时被刷新。

9.3.6 优先级直接中断处理

优先级直接中断处理与优先级标准中断处理的区别之一是: 一些操作被移出中断入口处理程序, 而放到各自的 ISR 中。被移出的包括屏蔽低优先级中断的代码。每个 ISR 都将屏蔽掉相对于自身的低优先级中断, 由于预先设定了每个中断的优先级, 所以优先级是一个固定的值。

之二是: 优先级直接中断处理程序直接跳转到合适的 ISR。每个 ISR 在修改 cpsr 重新允许中断前, 要禁止低优先级的中断。这种类型的处理程序头部相对简单, 因为屏蔽工作是由各自的 ISR 完成的。但这样会带来少量的重复代码, 因为每一个中断服务例程都必须完成这一任务。

【例 9.13】 bit_x 定义把一个中断源与中断控制器中的某一位对应起来, 用以一个 ISR 内屏蔽低优先级的中断。

保存好上下文后, ISR 表的基地址就被装载到寄存器 r12。一旦中断源的优先级被确定, 使用寄存器 r12 就可以跳转到正确的 ISR。

I_Bit	EQU	0x80	
PRIORITY_0	EQU	2	;通信接收
PRIORITY_1	EQU	1	;通信发送
PRIORITY_2	EQU	0	;定时器 1
PRIORITY_3	EQU	3	;定时器 2
BINARY_0	EQU	1 << PRIORITY_0	;1 << 2 0x00000004
BINARY_1	EQU	1 << PRIORITY_1	;1 << 1 0x00000002
BINARY_2	EQU	1 << PRIORITY_2	;1 << 0 0x00000001
BINARY_3	EQU	1 << PRIORITY_3	;1 << 3 0x00000008
MASK_3	EQU	BINARY_3	
MASK_2	EQU	MASK_3 + BINARY_2	
MASK_1	EQU	MASK_2 + BINARY_1	
MASK_0	EQU	MASK_1 + BINARY_0	
ic_Base	EQU	0x80000000	
IRQStatus	EQU	0x0	
IRQRawStatus	EQU	0x4	
IRQEnable	EQU	0x8	
IRQEnableSet	EQU	0x8	
IRQEnableClear	EQU	0xc	
bit_timer1	EQU	0	
bit_commtx	EQU	1	
bit_commr	EQU	2	
bit_timer2	EQU	3	
IRQ_Handler	;指令		注释
SUB	r14,r14, #4		;r14_irq -= 4
STMF	r13!,{r14}		;保存 r14_irq
MRS	r14,spsr		;复制 spsr_irq
STMF	r13!,{r10,r11,r12,r14}		;保护上下文
LDR	r14, = ic_Base		;中断控制器地址
LDR	r10,[r14, #IRQStatus]		;装载 IRQ 状态
ADR	r12,isr_table		;得到 ISR 表地址

```

TST      r10, # BINARY_0           ;如果中断源为通信接收
LDRNE    pc, [r12, # PRIORITY_0 << 2] ;则跳转到通信接收 ISR
TST      r10, # BINARY_1           ;如果中断源为通信发送
LDRNE    pc, [r12, # PRIORITY_1 << 2] ;则跳转到通信发送 ISR
TST      r10, # BINARY_2           ;如果中断源为定时器 1
LDRNE    pc, [r12, # PRIORITY_2 << 2] ;则跳转到定时器 1 ISR
TST      r10, # BINARY_3           ;如果中断源为定时器 2
LDRNE    pc, [r12, # PRIORITY_3 << 2] ;则跳转到定时器 2 ISR
B        service_none

isr_table
DCD      service_timer1           ;定时器 1 ISR
DCD      service_commtx           ;通信发送 ISR
DCD      service_commr           ;通信接收 ISR
DCD      service_timer2           ;定时器 2 ISR

priority_masks
DCD      MASK_2                   ;优先级屏蔽码 2
DCD      MASK_1                   ;优先级屏蔽码 1
DCD      MASK_0                   ;优先级屏蔽码 0
DCD      MASK_3                   ;优先级屏蔽码 3

service_timer1
MOV      r11, # bit_timer1        ;复制 bit_timer1
LDR      r14, = ic_Base            ;中断控制器地址
LDR      r12, [r14, # IRQEnable]   ;IRQ 允许寄存器
ADR      r10, priority_masks       ;得到地址优先级地址
LDR      r10, [r10, r11, LSL # 2]   ;装载优先级屏蔽码
AND      r12, r12, r10             ;AND 允许寄存器
STR      r12, [r14, # IRQEnableClear] ;清除中断
MRS      r14, cpsr                 ;复制 cpsr
BIC      r14, r14, # I_Bit          ;清零 I 位
MSR      cpsr_c, r14               ;允许 IRQ

```

<ISR 的其余部分>

优先级中断的确定是通过从最高优先级中断开始,向下直到最低优先级中断的检查来实现的。优先级中断一旦确定,就把合适的 ISR 地址装载到 pc,即 isr_table 中保存的间接地址加上优先级左移 2 位(即乘以 4)。或者可以使用条件分支指令 BNE 来跳转到合适

的 ISR。

ISR 跳转表 `isr_table` 是以最高优先级的中断为开始排序的。

`service_timer1` 入口部分说明了优先级直接中断处理程序使用的 ISR 例子。每个 ISR 都是惟一的,依赖于特定的中断源。

中断控制器的基地址复制到寄存器 `r14_irq`。这个地址加上一个偏移量用来复制 `IRQEnable` 寄存器的内容到寄存器 `r12`。

优先级屏蔽码表的地址应复制到寄存器 `r10`,以便用来计算实际屏蔽码的地址。寄存器 `r11` 左移 2 位,产生 0,4,8 或 12 字节的偏移。这个偏移量加上优先级屏蔽码表的基地址,就可以把屏蔽码装载到寄存器 `r10`。这里的优先级屏蔽码表与上一节的相同。

寄存器 `r10` 包含了 ISR 屏蔽码,寄存器 `r12` 包含了当前屏蔽码,用一个二进制“与”来合并这 2 个屏蔽码。然后使用新的屏蔽码,通过 `IRQEnableClear` 寄存器来配置中断控制器。现在通过清除 `cpsr` 里的 *i* 位来允许 IRQ 异常就安全了。

处理程序可以继续为当前的中断服务,除非有更高优先级的中断发生,那样的话,高优先级的中断将打断当前的中断服务。

小 结 优先级直接中断处理

- 在更短的时间内处理高优先级的中断,直接跳转到特定的 ISR;
- 低的中断延迟;
- 优点:使用简单的跳转,为进入 ISR 节省了宝贵的时间;
- 缺点:每个 ISR 都需要设置外部中断屏蔽码,以阻止低优先级中断打断当前的 ISR,这使每个 ISR 增加了额外的代码。

9.3.7 优先级分组中断处理

优先级分组中断处理与其它优先级中断处理不同,它是针对大量的中断集而设计的,是通过把中断分组并形成具有同一优先级的子集来实现的。

一个嵌入式系统的设计者必须明确每个子集的中断源,并为该子集赋予一个组优先级。由于各组决定了系统的特性,在为中断源进行选择分组时需谨慎,这一点很重要。把中断源集中管理,相对降低了处理程序的复杂度,因为没有必要依次检查每个中断,以确定优先级。如果一个优先级分组中断处理程序设计得好,则可大大改善系统整体的响应时间。

【例 9.14】 针对 2 个优先级组设计的处理程序。

定时器中断源被划分到组 0,通信中断源被划分到组 1(见表 9.11)。组 0 的中断优先级比组 1 的高。

表 9.11 分组中断源

分 组	中 断
0	定时器 1, 定时器 2
1	通信发送, 通信接收

```

I_Bit                EQU 0x80

PRIORITY_0            EQU    2                ;通信接收
PRIORITY_1            EQU    1                ;通信发送
PRIORITY_2            EQU    0                ;定时器 1
PRIORITY_3            EQU    3                ;定时器 2

BINARY_0              EQU    1 << PRIORITY_0    ;1 << 2 0x00000004
BINARY_1              EQU    1 << PRIORITY_1    ;1 << 1 0x00000002
BINARY_2              EQU    1 << PRIORITY_2    ;1 << 0 0x00000001
BINARY_3              EQU    1 << PRIORITY_3    ;1 << 3 0x00000008

GROUP_0               EQU    BINARY_2 | BINARY_3
GROUP_1               EQU    BINARY_0 | BINARY_1

GMASK_1               EQU    GROUP_1
GMASK_0               EQU    GMASK_1 + GROUP_0

MASK_TIMER1           EQU    GMASK_0
MASK_COMMTX           EQU    GMASK_1
MASK_COMMRX           EQU    GMASK_1
MASK_TIMER2           EQU    GMASK_0

ic_Base               EQU    0x80000000
IRQStatus              EQU    0x0
IRQRawStatus          EQU    0x4
IRQEnable             EQU    0x8
IRQEnableSet          EQU    0x8
IRQEnableClear        EQU    0xc

Interrupt_handler
    
```

```

SUB      r14,r14,#4           ;r14_irq-=4
STMFD    r13!,{r14}          ;保存 r14_irq
MRS      r14,spsr             ;复制 spsr_irq
STMFD    r13!,{r10,r11,r12,r14} ;保护上下文
LDR      r14,=ic_Base         ;中断控制器地址
LDR      r10,[r14,#IRQStatus] ;装载 IRQ 状态
ANDS     r11,r10,#GROUP_0     ;属于 GROUP_0
ANDEQS   r11,r10,#GROUP_1     ;属于 GROUP_1
AND      r10,r11,#0xf         ;屏蔽高 24 位
ADR      r11,lowest_significant_bit ;装载 LSB 地址
LDRB     r11,[r11,r10]        ;装载字节
B        disable_lower_priority ;跳转到例程

```

lowest_significant_bit

```

;      0  1 2 3 4 5 6 7 8 9 a b c d e f
DCB     0xff,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0

```

disable_lower_priority

```

CMP      r11,#0xff            ;如果未知
BEQ      unknown_condition    ;则跳转
LDR      r12,[r14,#IRQEnable] ;装载 IRQ 允许寄存器
ADR      r10,priority_mask     ;装载优先级地址
LDR      r10,[r10,r11,LSL #2]  ;mem32[r10+r11 << 2]
AND      r12,r12,r10           ;AND 允许寄存器
STR      r12,[r14,#IRQEnableClear] ;清除新中断
MRS      r14,cpsr              ;复制 cpsr
BIC      r14,r14,#I_Bit        ;清零 I 位
MSR      cpsr_c,r14            ;允许 IRQ 中断
LDR      pc,[pc,r11,LSL #2]    ;跳转到一个 ISR
NOP
DCD      service_timer1        ;定时器 1 ISR
DCD      service_commtx        ;通信发送 ISR
DCD      service_commr        ;通信接收 ISR
DCD      service_timer2        ;定时器 2 ISR

```

priority_mask

```

DCD      MASK_TIMER1           ;mask GROUP 0

```

ARM 嵌入式系统开发

```

DCD      MASK_COMMTX      ;mask GROUP 1
DCD      MASK_COMMRX      ;mask GROUP 1
DCD      MASK_TIMER2      ;mask GROUP 0

```

通过使用二进制“或”操作, GROUP_*x* 定义为各中断源赋予了它们特定的优先级。GMASK_*x* 定义为分组的中断指定了屏蔽码。MASK_*x* 定义使每一个 GMASK_*x* 对应于一个特定的中断源,后面它们将被用在优先级屏蔽码表中。

保护好上下文后,处理程序就读取 IRQ 状态寄存器,其地址是中断控制器的基地址加上一个偏移量。

然后,处理程序使用逻辑“与”来确认中断源属于哪一组。指令的后缀字母 S 表明要更新 cpsr 中的条件标志位。

寄存器 r11 现在保存的是优先级最高的分组 0 或 1。处理程序把 r11 和 0xf 进行逻辑“与”来屏蔽其它中断。

接着,最低有效位表的地址被装载到寄存器 r11。使用寄存器 r10 的值(0,1,2 或 3)从表格起始装载 1 字节(见表 9.12)。一旦最低有效位的位置装载到寄存器 r11,处理程序就跳转到一个例程。

表 9.12 最低有效位表

二进制格式	值	二进制格式	值
0000	未知	1000	3
0001	0	1001	0
0010	1	1010	1
0011	0	1011	0
0100	2	1100	2
0101	0	1101	0
0110	1	1110	1
0111	0	1111	0

disable_lower_priority 中断例程首先检查是否有无效的(spurious)(或不再存在的)中断。如果这个中断已经失效,则调用 unknown_condition 例程。处理程序装载 IRQEnable 寄存器,并把结果放在寄存器 r12 里。

以装载优先级屏蔽码表的地址得到优先级屏蔽码,然后左移 2 位寄存器 r11 里的数据。结果 0,4,8 或 12 加到优先级屏蔽码的地址。寄存器 r10 保存的就是禁止低优先级组中断发生的屏蔽码。

下一步是清除低优先级的中断,使用这个屏蔽码与寄存器 r12(IRQEnable 寄存器)相

“与”，清零对应的位；然后把结果保存到 IRQEnableClear 寄存器。现在清除 cpsr 里的 i 位来允许 IRQ 异常就安全了。

最后，处理程序通过修改寄存器 r11（它仍然保存有最高优先级中断）和 pc 跳转到正确的中断服务例程。寄存器 r11 左移 2 位，并把结果加到 pc，ISR 的地址就确定了。这一地址直接装载到 pc。

注意：跳转表必须跟在 LDR 指令后。因为 ARM 的流水线机制需要插入一个 NOP。

小结 优先级分组中断处理

- 这是一种处理不同优先级组别中断的方法；
- 低的中断延迟；
- 优点：当系统要处理大量中断时很有用，同样缩短了响应时间，因为确定优先级的时间较短；
- 缺点：须决定中断如何分组。

9.3.8 基于 VIC PL190 的中断服务例程

为了利用向量中断控制器的优点，IRQ 中断向量入口作了修改。

```
0x00000018    LDR    pc,[pc,# - 0xff0] ;IRQ pc = mem32[0xfffff030]
```

该指令从内存映射地址 0xfffff030 中获得数据装载到 pc，这就跳过了任何软件中断处理，因为可以直接从硬件获得中断源。由于只使用了一句简单的跳转指令，这样做也减少了中断延迟。

以下是一个 VIC 服务程序的例子：

```
INTON          EQU    0x0000          ;允许中断
SYS32md        EQU    0x1f            ;系统模式
IRQ32md        EQU    0x12            ;IRQ 模式
I_Bit          EQU    0x80
VICBaseAddr    EQU    0xfffff000      ;VIC 控制器基地址
VICVectorAddr  EQU    VICBaseAddr + 0x30 ;中断的 ISR 地址
```

vector_service_routine

```
    SUB        r14,r14,#4              ;r14 -= 4
    STMFED     r13!,{r0 - r3,r12,r14} ;保护上下文
    MRS        r12,spsr                ;复制 spsr
    STMFED     r13!,{r12}              ;保存 spsr
    ;<清除中断源>
```

```

MSR      cpsr_c, # I_NTON|SYS32md      ;cpsr_c = ift_sys
; <中断服务代码 >
MSR      cpsr_c, # I_Bit|IRQ32md      ;cpsr_c = Ift_irq
LDMFD    r13!, {r12}                  ;恢复 (spsr_irq)
MSR      spsr_cxsf, r12                ;恢复 spsr
LDR      r1, = VICVectorAddr          ;装载向量地址
STR      r0, [r1]                     ;服务完成
LDMFD    r13!, {r0 - r3, r12, pc}^    ;返回

```

该例程在清除中断源之前保存了上下文及 `spsr_irq`。一旦完成该操作,就可以通过清除 `i` 位,使 IRQ 异常重新使能,并将处理器模式设为系统模式。于是服务程序在系统模式下处理中断。一旦完成处理,设置 `i` 位禁止 IRQ 异常,并将处理器的模式切换回 IRQ 模式。

`spsr_irq` 也从 IRQ 堆栈中被恢复,为返回做准备。

然后,中断服务程序要对控制器中的 `VICVectorAddr` 寄存器进行写。写这个寄存器的目的是,向优先级处理硬件表明,该中断已经被服务过。

注意: VIC 基本上是一种硬件中断控制器,因此必须在 VIC 被激活之前,编程和设置好中断服务程序的地址列表。

9.4 总 结

异常会改变指令执行的正常顺序。ARM 有 7 种异常:数据中止、快速中断请求、中断请求、预取指中止、软件中止、复位和未定义指令中止。每一种异常与 ARM 处理器的一种模式相对应。一旦异常发生,处理器便进入一种特定的模式,并跳转到向量表中的某个入口。每种异常也有一个优先级。

中断是由 ARM 外设引起的一种特殊的异常。IRQ 异常用于通常的操作系统事务处理。FIQ 异常一般是为单独的中断源保留的。*中断延迟* 是指从外部中断请求信号出现,到特定的中断服务程序 (ISR) 的第一条指令被取指之间的时间间隔。

本章讨论了 8 种中断处理方法,从依次服务单个中断、非常简单的无嵌套中断处理,到把中断划分到不同优先级组别的高级优先级分组中断处理,还有基于向量中断控制器 (VIC) 的中断服务例程。

第 10 章

固 件

- 固件和引导装载程序
- 例子: Sandstone
- 总 结

本章讨论基于 ARM 的嵌入式系统的固件(Firmware)。对于任何嵌入式系统,固件都是一个重要的部分,因为在一个新的平台上,固件通常是被移植并执行的第一段代码。对不同的系统,固件有很大的差异:可以是一个完整的嵌入式软件系统,也可以只是一段简单的初始化和引导装载(bootloader)程序。本章将分为 2 节:

10.1 节介绍固件。这一节将给固件下一个定义,并介绍 2 个可以在 ARM 处理器上使用的流行的工业标准固件包——ARM Firmware Suite 和 Red Hat 公司的 RedBoot。这 2 个固件包是通用的,可以较容易和快速地移植到不同的 ARM 平台上。

10.2 节重点讨论初始化和引导装载过程。为了讨论方便,本节介绍了一个简单的例子——Sandstone。Sandstone 首先初始化硬件,然后装载一个映像文件(Image)到存储器,最后将 pc 指针的控制权交给该映像文件。

接下来讨论固件并介绍 2 个通用的 ARM 固件包。

10.1 固件和引导装载程序

有些术语,不同的人往往会有不同的认识,本章使用如下定义。

- **固件** 是底层的嵌入式软件,它提供硬件和应用程序/操作系统层软件之间的接口。固件存储在 ROM 里,嵌入式硬件系统一上电就立即执行。在完成系统初始化以后,固件可以继续保持活动状态,以提供某些基本的系统操作。对于一个基于 ARM 的系统,选择什么固件取决于特定的应用:可以是装载并执行一个复杂的操作系统,也可以只是简单地将控制权交给一个小的微内核。因此,固件实现的需求会有很大的不同。例如一个小的系统可能只需要一个最小的固件支持,用来引导一个小的操作系统。固件的一个主要目的是,提供一种可靠的机制来装载和引导一个操作系统。
- **引导装载程序(Bootloader)** 是一个用来引导操作系统或应用程序到硬件目标平台上的小应用程序,它在操作系统或应用程序执行以后便立即退出。引导装载程序通常包含在固件里。

为了有助于理解不同固件的实现要素,这里列出一个通常的固件执行流程(见表 10.1)。下面将详细讨论每一个执行阶段。

第 1 阶段是设置目标平台——准备一个操作系统引导时所需的环境,因为操作系统在运行前都需要一个特定的环境。这一阶段包括正确地初始化平台(例如,要保证某个特定微控制器的控制寄存器已经被赋予恰当的地址;或者通过改变存储器映射,得到一个期望的存储器结构)。

表 10.1 固件执行流程

执行阶段	特 征
设置目标平台	编程硬件系统寄存器 平台识别 诊断 调试接口 命令行解释器
抽象硬件	硬件抽象层 设备驱动
装载可引导的映像文件	基本的文件系统
交出控制权	改变 pc 指针,使它指向新的映像文件

同一段可执行代码经常需要在不同的内核和平台上运行,在这种情况下,固件必须能够识别它正运行在何种内核和平台上。内核的识别通常只须读取协处理器 cp15 的寄存器 r0,其中保存有处理器的型号和生产商的名字。有多种方法可以用来识别平台,例如检查一组特定的外设是否存在,或简单地读取一个可预编程的芯片内容。

诊断软件提供一种有效的方法来快速检测出一些基本的硬件故障。由于它是用来检测硬件的,因此诊断软件都与特定的硬件有关。

调试功能是以模块(module)或监视器(monitor)的形式提供的,它为调试运行在硬件目标平台上的代码提供软件支持,这些支持包括:

- 在 RAM 中建立断点,断点允许中断程序和查看处理器内核的状态;
- 列出、修改存储器的值(使用 peek 和 poke 操作);
- 显示当前处理器寄存器的内容;
- 将存储器内容反汇编成 ARM 和 Thumb 指令。

交互功能:可以通过命令行解释器 CLI(Command Line Interpreter)或者与目标平台相连的专用主机调试器来发送命令。除非固件可以访问内部硬件调试电路,否则只有在 RAM 中的映像文件可以通过软件调试机制来进行调试。

命令行解释器 CLI 通常在较高级的固件实现中才有。可以通过在命令行提示符后面键入命令来更改默认配置,从而改变将要引导的操作系统。对于嵌入式系统,CLI 通常需要通过一个主机终端应用程序来控制。主机和目标平台之间的通信一般通过串口或网络接口。

第 2 阶段是抽象硬件。硬件抽象层 HAL(Hardware Abstraction Layer)是一个软件层,它向上通过提供一组已定义的编程接口来隐藏下层的硬件。当移植到一个新的目标平

ARM 嵌入式系统开发

台时,这些编程接口保持不变,但下层的实现却改变了。例如,2个目标平台可能使用不同的时钟外设,每个外设的初始化和配置都需要新的代码,即使硬件和软件在实现上有很大不同,HAL 编程接口都将保持不变。

HAL 中与特定硬件外设通信的软件称为设备驱动(device driver)。每个设备驱动提供一个标准的应用程序编程接口(API)来对特定外设进行读/写。

第3阶段是装载一个可引导的映像文件。是否要实现这个功能,取决于用来存储映像文件的存储媒介。

注意:并不是所有的操作系统映像文件或应用程序映像文件都需要拷贝到 RAM 中,它们也可以简单地直接在 ROM 中执行。

ARM 处理器通常都在一个包含 Flash ROM 的小设备上,一般都带有一个简单的 Flash ROM 文件系统(FFS),允许存储多个可执行的映像文件。

其它的存储媒介,比如硬盘,需要固件中包含能够访问该存储媒介的设备驱动。访问该硬件时,固件需要知道底层文件系统的格式,这样固件才能访问文件系统,找到包含映像的文件,然后将其复制到内存。类似的,如果映像文件是在网络上的,固件就需要知道网络协议和以太网硬件。

装载过程中必须考虑映像文件的格式。最基本的映像文件格式是普通的二进制格式,这种格式的映像文件不包含任何头部或调试信息。在基于 ARM 的系统中,一种常用的映像文件格式是可执行和连接格式 ELF(Executable and Linking Format)。这种格式最初是为 UNIX 系统开发的,用来代替早期的普通对象文件格式 COFF(Common Object File Format)。ELF 文件有 3 种形式:可重定位(relocatable)、可执行(executable)和共享对象(shared object)。

大多数固件系统都必须处理可执行的格式。装载一个 ELF 映像文件包括要解释标准的 ELF 头部信息(执行地址、类型、文件大小等)。映像文件也可能经过加密或压缩,这样的话,装载过程还包括执行解密或解压工作。

第4阶段是转交控制权。这个阶段,固件将平台的控制权交给操作系统或应用程序。

注意:并不是所有的固件在这个阶段都交出控制权,固件也可以在平台上保留控制软件。

如果固件将控制权交给操作系统,则在操作系统取得控制权后,固件一般就处于非活动状态。固件的机器无关层 MIL(Machine Independent Layer)或者硬件抽象层 HAL(Hardware Abstraction Layer)部分固件也可以继续保持活动状态,这一层通过 SWI(软中断)机制为特定的硬件设备提供一个标准的应用程序接口。

在 ARM 系统中,交出控制权就是更新向量表和修改 pc 指针。更新向量表包括修改特定的异常和中断向量,使其指向操作系统中用来处理该异常或中断的处理程序。pc 指针必须被修改,使其指向操作系统的入口地址。

在一些比较成熟的操作系统中,比如 Linux,转交控制权需要传递给操作系统内核(kernel)一个标准的数据结构,这个数据结构说明了操作系统内核将运行的环境。例如,数据结构中可能有一个域包含目标平台的可用 RAM 的大小,另一个域则可能包含所用 MMU 的类型。

接下来应用以上的定义来描述 2 个通用的固件套件(firmware suite)。

10.1.1 ARM Firmware Suite

ARM Firmware Suite(AFS)是 ARM 公司专门为基于 ARM 的嵌入式系统开发的固件包,它支持包括 Intel 公司的 XScale 和 StrongARM 处理器在内的很多 ARM 处理器和平台。该固件包主要包括 2 项技术:硬件抽象层 μ HAL(发音为“micro-HAL”)和调试监控 Angel。

μ HAL 提供了一个可以在不同通信设备(例如 USB、以太网、串口等)上操作的底层设备驱动框架,同时它也提供标准的应用程序编程接口(API)。这样,在移植代码时,只要实现与 μ HAL 的 API 函数相对应的、与具体硬件相关的部分即可。

使用标准的函数编程接口,可以使移植过程变得相对容易。一旦在新的目标平台移植好了固件,接着就是移植一个操作系统到这个平台。移植过程的复杂性取决于该 OS 是否使用了 μ HAL 的 API 调用来访问硬件。

μ HAL 主要支持如下特征:

- **系统初始化**——设置目标平台和处理器内核,这可以是一个简单、也可以是一个复杂的任务,取决于目标平台的复杂性;
- **简单的串口驱动**——提供一种基本的与主机进行通信的方法;
- **LED 支持**——控制 LED,以实现简单的用户反馈(feedback),使应用程序可以显示一些操作状态;
- **定时器支持**——产生周期性的定时中断,这是抢占式调度(preemptive context switching)操作系统实现其调度机制所必需的;
- **中断控制器**——支持不同的中断控制器。

在 μ HAL 的启动监控中,包含一个命令行解释器 CLI。

Angel 允许调试的主机和目标平台之间的通信,使程序员可以查看和修改存储器内容,下载和执行映像文件,设置断点,查看处理器寄存器的内容。所有这些控制都通过主机调试器。Angel 调试监控必须可以访问 SWI 和 IRQ 或 FIQ 中断向量。

Angel 使用 SWI 指令来提供一组 API 函数,允许一个程序打开、读/写主机的文件系统。IRQ/FIQ 中断用于与主机的通信。

10.1.2 Red Hat Redboot

RedBoot 是由 Red Hat 公司开发的固件工具,遵循开放源码许可(open source license),没有版税或前期费用。RedBoot 可以在不同的 CPU(如 ARM, MIPS, SH 等)上执行。它提供了 GNU 的 GDB 和 bootloader 两种调试方法。RedBoot 的软件核心基于一个 HAL。

RedBoot 主要支持的特征如下。

- **通信** 通过串口或以太网进行配置。对于串口,使用 X-Modem 协议与 GNU 调试器(GDB)进行通信;对于以太网,使用 TCP 协议与 GDB 进行通信。RedBoot 支持很多网络标准,如 bootp, telnet 和 tftp。
- **Flash 存储器管理** 提供一组文件系统函数,用它可以下载、升级或擦除 Flash 存储器里的映像文件。另外,映像可以被压缩或解压。
- **完全的操作系统支持** 支持嵌入式 Linux、Red Hat 的 eCos 和其它许多常见操作系统的装载和引导。对于嵌入式 Linux, RedBoot 支持定义一些参数,这些参数在引导时直接传给操作系统内核。

10.2 例子:Sandstone

358

本节设计的 Sandstone 是一个最小的系统,只执行如下任务:设置目标平台环境,装载一个可引导的映像到存储器,将控制权交给操作系统。但它是一个真实运行的例子。

表 10.2 Sandstone 概述

属 性	配 置
代码	只使用 ARM 指令
工具链	ADS1.2
映像大小	700 字节
源代码	17 KB
存储器	重映射过

这个实现针对 ARM Evaluator-7T 评估板,它包含一个 ARM7TDMI 的处理器。这个例子清楚地说明了如何设置一个简单的平台,以及软件是如何被装载到存储器中并被引导的。被装载的软件既可以是一个应用程序,也可以是一个操作系统映像。Sandstone 是一个静态的设计,在编译以后不能再配置。表 10.2 列出了 Sandstone 的基本特征。

下面再来看一下目录和代码结构。目录结构显示了源代码放在哪里,以及不同的编译文件放在哪里。代码结构着重介绍实际的初始化和引导过程。

请注意 Sandstone 是完全用 ARM 汇编写的,它是可以初始化目标硬件和引导任何软件的一段正常工作代码。当然,是指在 ARM Evaluator-7T 评估板上。

10.2.1 Sandstone 的目录结构

Sandstone 可以在作者网页上找到。其目录结构如图 10.1 所示。这个结构所遵循的标准类型在以后的章节中将继续使用。Sandstone 源文件 `sand.s` 位于 `sand/build/src` 目录中。

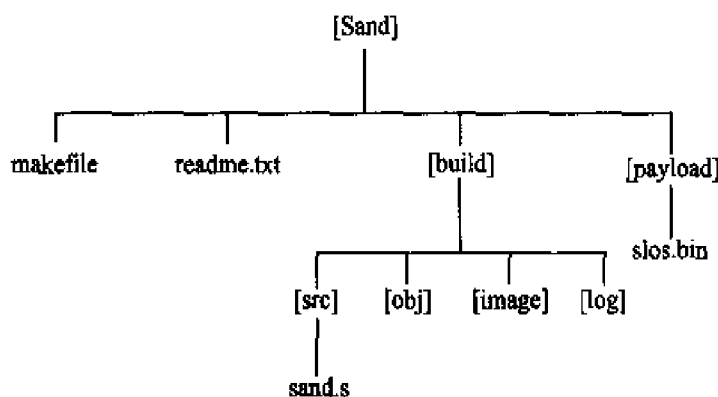


图 10.1 Sandstone 目录结构

汇编器生成的目标文件放在 `build/obj` 目录下。目标文件被连接后,最后的 Sandstone 映像文件放在 `sand/build/image` 目录下,这个映像文件包括 Sandstone 代码和有效载荷(payload)2 部分。将被 Sandstone 装载并引导的映像文件(有效载荷)放在 `sand/payload` 目录下。

关于 Sandstone 的 build 过程的详细介绍,可以参看 `sand` 目录下的 `readme.txt` 文件。`readme.txt` 包含如何生成一个在 ARM Evaluator-7T 上运行的二进制映像文件的介绍。

10.2.2 Sandstone 的代码结构

Sandstone 只包含一个汇编文件。文件结构被分成几步,每一步对应 Sandstone 的执行流程的一个阶段(见表 10.3)。

下面将详细介绍这些步骤。这里尽量避免与平台相关的部分,但是有些与硬件相关的部分是不可避免的(如配置系统的寄存器和存储器重映射)。

Sandstone 初始化工作的目标是设置目标平台环境,这样它就可提供某些形式的反馈(如 LED)来指示固件正在执行和控制着平台。

表 10.3 Sandstone 执行流程

步	描述
1	执行复位异常
2	开始初始化硬件
3	存储器重映射
4	初始化通信硬件
5	引导装载——复制映像文件和转交控制权

10.2.2.1 第 1 步: 执行复位异常

指令执行从复位异常开始。默认的向量表中只要有一个复位向量入口,它就是真正被执行的第 1 条指令。除了复位向量外的其它向量都跳转到一个哑处理过程——一条分支指令跳转到一个无限循环。这里假设在 Sandstone 的执行过程中没有异常或中断发生。复位向量使执行流程进入第 2 步。

```

AREA start, CODE, READONLY
ENTRY

sandstone_start
    B    sandstone_init1    ;复位向量
    B    ex_und              ;未定义向量
    B    ex_swi              ;swi 向量
    B    ex_pabt             ;预取指中止向量
    B    ex_dabt             ;数据中止向量
    NOP                      ;未用
    B    int_irq             ;IRQ 向量
    B    int_fiq             ;FIQ 向量

ex_und    B    ex_und        ;死循环
ex_swi    B    ex_swi        ;死循环
ex_dabt   B    ex_dabt       ;死循环
ex_pabt   B    ex_pabt       ;死循环
int_irq   B    int_irq       ;死循环
int_fiq   B    int_fiq       ;死循环

```

Sandstone_start 被放在 0x00000000。

第 1 步的执行结果:

- 哑处理过程被设置;
- 控制权被转交给初始化硬件的代码。

10.2.2.2 第 2 步: 开始初始化硬件

初始化硬件的主要工作是设置系统寄存器。访问硬件设备之前,必须先初始化这些寄存器。例如,ARM Evaluator-7T 评估板有一个 7 段数码管显示器,可以把它作为反馈工具,用来指示固件正被执行。在显示段码之前,必须将系统寄存器的基地址放在一个已知的位置。这里选取默认地址 0x03ff0000,这样可以使所有的硬件系统寄存器远离 ROM 和 RAM,区分了外设和存储器地址空间。

这样,所有微控制器的存储器映射寄存器将被放在从地址 0x03ff0000 开始的特定偏移处。该操作可用下面的代码来实现:

```
sandstone_init1
    LDR    r3, = SYSCFG      ;这里 SYSCFG = 0x03ff0000
    LDR    r4, = 0x03fffa0
    STR    r4,[r3]
```

寄存器 r3 包含默认的系统寄存器基地址,并可被用来设置新的默认地址,也可用来设置其它类似于 cache 的特定属性。寄存器 r4 包含新的配置值,其高 16 位包含新的系统寄存器基地址的高地址 0x03ff,低 16 位包含新的属性配置值 0xffa0。

设置完系统寄存器的基地址后,就可以开始配置数码管显示了。用数码管来指示 Sandstone 的执行过程。这里没有描述数码管显示的代码,因为它是与具体硬件相关的。

第 2 步的执行结果:

- 系统寄存器被设置在一个已知的基地址——0x03ff0000;
- 数码管显示器被配置,这样就可以用它来显示执行过程。

10.2.2.3 第 3 步:存储器重映射

硬件初始化的一个主要任务是设置存储器环境。Sandstone 初始化 SRAM,然后进行存储器重映射。这个过程应在系统初始化时尽早完成。在一个已知的存储器状态下,平台开始运行。如表 10.4 所列。

表 10.4 初始存储器状态

存储器类型	起始地址	结束地址	大小/KB
Flash ROM	0x00000000	0x00080000	512
SRAM bank 0	不可用	不可用	256
SRAM bank 1	不可用	不可用	256

可以看到,硬件平台上电时,只有 Flash ROM 被分配在存储器映射的一个位置,2 个 SRAM 段(bank 0 和 bank1)没有初始化,所以还不能使用。下一步就是使 2 个 SRAM 段可用,并将 Flash ROM 重映射到一个新的地址。实现代码如下:

```
LDR    r14, = sandstone_init2
LDR    r4, = 0x01800000      ;新的 Flash ROM 位置
ADD    r14,r14,r4
ADRL   r0, memorymaptable_str
LDMIA  r0, {r1 - r12}
```

ARM 嵌入式系统开发

```

LDR    r0, = EXTDBWTH          ; = (SYSCFG + 0x3010)
STMIA  r0, {r1 - r12}
MOV    pc, r14                  ; 跳转到重映射后的存储空间

```

sandstone_init2

; sandstone_init2 之后的代码执行在 @ + 0x01800000

在重映射之前,代码的第 1 部分是计算 sandstone_init2 的绝对地址。Sandstone 用该地址跳转到重映射以后新环境的下一个例程。

代码的第 2 部分执行存储器重映射。新的存储器映射数据从一个 memorymapable_str 指向的结构被装载到寄存器 r1~r12。然后,使用这些寄存器,该结构被写到从系统配置寄存器偏移 0x3010 的存储器控制器。这一步执行完成后,新的存储器映射就建立了(见表 10.5)。

表 10.5 重映射以后

类 型	起始地址	结束地址	大小/KB
Flash ROM	0x01800000	0x01880000	512
SRAM bank 0	0x00000000	0x00040000	256
SRAM bank 1	0x00040000	0x00080000	256

可以看到,SRAM 现在可使用了,Flash ROM 被放置在一个更高的地址上。最后的部分是跳转到固件的下一个例程或阶段。

这个跳转是利用 ARM 流水线来实现的。尽管新的存储器环境激活了,但下一条指令已经装载到流水线里了。下一个例程可以通过将寄存器 r14 的值复制到 pc 而被调用。这里在重映射代码的后面紧跟一条简单的 MOV 指令来实现这一功能。

第 3 步的执行结果:

- 存储器被重映射,如表 10.5 所列;
- pc 指向下一步,这个地址在重映射后的 Flash ROM 内。

10.2.2.4 第 4 步:初始化通信硬件

通信初始化包括配置串口和输出一个标准的提示符。显示出提示符,表示固件已经全部正常工作且存储器已成功重映射。同样,由于在 ARM 评估板 Evaluator - 7T 上初始化串口的代码是与硬件相关的,这里就不列出了。

串口被设置成 9 600 波特,无校验,1 位停止位,无数据流控制。如果串口电缆接到评估板上,则主机终端就必须用上面相同的配置。

第 4 步的执行结果:

- 串口被初始化——9 600 波特,无奇偶校验,1 位停止位,无数据流控制。
- 串口输出 Sandstone 的提示符:

Standstone Firmware (0.01)

```
- platform ..... e7t
- status ..... alive
- memory ..... remapped
```

```
+ booting payload ...
```

10.2.2.5 第 5 步:引导装载——复制映像文件和转交控制权

最后一步包括复制一个有效载荷(映像文件),并将 pc 的控制权转交给该程序。这项工作通过下面的代码来实现。代码的第一部分初始化作为块复制工作寄存器的 r12, r13 和 r14。引导装载代码假设该有效载荷是一个纯二进制的映像文件,无须解密或解压。

```
sandstone_load_and_boot
    MOV        r13, #0                ;目标地址
    LDR        r12, payload_start_address ;起始地址
    LDR        r14, payload_end_address  ;结束地址
_copy
    LDMIA      r12!, {r0 - r11}
    STMIA      r13!, {r0 - r11}
    CMP        r12, r14
    BLE        _copy
    MOV        pc, #0
```

```
payload_start_address
    DCD        startAddress
payload_end_address
    DCD        endAddress
```

目标地址寄存器 r13 指向 SRAM 的起始地址,在这里为 0x00000000。源地址寄存器 r12 指向映像文件的起始地址,源地址寄存器 r14 指向映像文件的结束地址。使用这些地址,映像文件被复制到 SRAM。

通过强制把 pc 指向被复制的有效载荷的入口地址,就把 pc 的控制权转交给了该程序。对于这个例子,它的入口地址为 0x00000000。这样,被装载的程序就控制了系统。

第 5 步的执行结果:

- 有效载荷被复制到 SRAM,地址为 0x00000000。

ARM 嵌入式系统开发

- pc 指针的控制权转交给了被装载程序, $pc=0x00000000$ 。
- 系统完全被引导。下面的信息从串口输出:

Standstone Firmware (0.01)

```
- platform ..... e7t
- status ..... alive
- memory ..... remapped
```

```
+ booting payload ...
```

Simple Little OS (0.09)

```
- initialized ..... ok
- running on ..... e7t
```

e7t;

10.3 总 结

本章首先讨论了基于 ARM 的固件。固件是提供硬件和应用程序或操作系统接口的底层代码。引导程序(bootloader)装载操作系统或应用程序到存储器,然后将 pc 控制权交给被装载的软件。

其次,介绍了 ARM Firmware Suite 和 RedBoot。ARM Firmware Suite 是专门为基于 ARM 的系统设计的。RedBoot 更通用,可以用在其它的非 ARM 的处理器上。

最后介绍了一个固件例子 Sandstone。Sandstone 初始化硬件,然后以下列步骤装载引导一个映像文件:

- ① 执行复位异常操作;
- ② 开始初始化硬件,设置系统寄存器的基地址,初始化数码显示管;
- ③ 存储器重映射,ROM 地址=高地址,SRAM 地址= $0x00000000$;
- ④ 初始化通信硬件,使输出指向到串口;
- ⑤ 引导装载——装载一个可执行的映像文件(有效载荷)到 SRAM,将 pc 指针的控制权给它($pc=0x00000000$)。

这样就实现了 ARM7TDMI 嵌入式系统的全部初始化。

第 11 章

嵌入式操作系统

- 基本模块
- 实例：简单小型操作系统 SLOS
- 总 结

ARM 嵌入式系统开发

本章讨论嵌入式操作系统(OS)的实现。由于嵌入式操作系统是为某一特殊目的而设计的,因此它历来具有简单,实时性强,在有限的存储空间中运行等特点。随着嵌入式硬件的不断成熟,这些特点也不断发展、变化,传统上只能在桌面机 OS 上找到的一些特征,例如虚拟存储器,现在也已经移植进嵌入式系统。

操作系统涉及的范围很广,本章只论述组成嵌入式操作系统的基本模块。该操作系统建立在第 10 章介绍的固件例子的基础上。

本章分为两节;11.1 节简要介绍组成嵌入式操作系统的基本模块,同时指出一些针对 ARM 处理器的 OS 会涉及到的特定问题;11.2 节介绍一个操作系统的实例,称为简单小型操作系统 SLOS(Simple Little Operating System),SLOS 说明了操作系统基本模块的一个具体实现。

11.1 基本模块

一个操作系统由一些常见的底层模块组成,每个模块实现一个预定的功能。这些模块的相互作用和功能决定了这个特定操作系统的特征。

初始化代码(initialization)是操作系统执行的第一段代码,包括建立内部数据结构、全局变量和硬件环境。在固件把控制权交给操作系统时,初始化代码开始执行。操作系统的硬件初始化包括设置各种控制寄存器,初始化设备驱动程序。若操作系统为抢占式的,还须建立一个周期性的定时器中断。

存储器处理(memory handling)包括建立系统堆栈和任务堆栈。这些堆栈的位置决定了任务或系统可用的存储空间大小。系统堆栈的位置通常在操作系统初始化时设置。任务堆栈在何时建立,取决于该任务是静态的还是动态的。

静态任务在编译时被定义,并包含于操作系统映像中。这些任务的堆栈可在操作系统初始化时建立。作为例子,后面的 SLOS 就是一个基于静态任务的操作系统。

动态任务在装载并运行操作系统以后被装载和执行,它不是操作系统映像的一部分。这种任务的堆栈在创建任务时才建立(例如 Linux 中就是如此)。不同操作系统的存储器处理,其复杂程度也不同,取决于很多因素,比如所选择的 ARM 处理器核、微控制器的性能及最终目标硬件的物理存储器布局等。

11.2 节中将介绍的示例操作系统 SLOS 采用静态存储器。它简单配置微控制器内的寄存器,并设置各堆栈的位置。因为没有实现动态存储管理,所以找不到 malloc()和 free()函数的实现,这些函数一般可在标准 C 库中找到。

中断和异常处理的方法是操作系统结构设计的一部分。设计时必须考虑如何处理各种不同的异常:数据中止、快速中断请求、中断请求、预取中止、复位和软中断(SWI)等。

并非所有的异常都需要异常处理程序。比如,若目标板不使用 FIQ 中断,则无需 FIQ 中断处理程序。通常比较安全的做法是,为未使用的异常提供一个死循环,作为默认的异常处理程序。这种方法使得调试更容易:当系统不响应时,说明是在某一异常处理程序中死循环。同时这种方法可以使系统不受意外异常的影响。

像 SLOS 这种抢占式的操作系统,需要一个周期性的中断。一般情况下,这由目标硬件上的计数器或定时器产生。操作系统在初始化阶段设置该周期性中断的频率,通常只须给计数器或定时器的存储器映射寄存器赋一特定值即可。

当被激活时,计数器或定时器就开始递减该值,当值递减到零时,就产生一个中断,相应的 ISR 将处理这个中断。ISR 首先用一个新的起始值重新初始化该计数器或定时器,然后调用调度程序或其它专用例程。

相反,非抢占式的操作系统不需要周期性的中断,它使用不同的技术,例如轮询(*polling*)——不断检查设备状态的变化,如果状态改变,则执行相应的特定操作。

调度程序(*scheduler*)是决定下一个该执行哪个任务的算法。有很多可选的调度算法,最简单的一个称为循环算法(*round-robin algorithm*),它以固定的次序循环激活任务。调度算法的选择必须在效率、代码大小与复杂性之间折衷。

执行完调度程序后,新、老任务用上下文切换来实现交换。上下文切换将老任务的处理器寄存器数据保存到一个数据结构中,然后将新任务的数据装载到处理器寄存器中(此过程的细节请参见 11.2.6 小节)。

最后一个模块是驱动程序框架(*device driver framework*)——操作系统用于在不同硬件外设之间提供统一接口的机制。这个框架提供一种标准且简单的方法,把对特定外设的新支持集成到操作系统中。应用程序要访问特定外设时,必须有对应的可用驱动程序。框架必须为访问外设提供安全的方法(例如,不允许多于一个的应用程序同时访问同一个外设)。

11.2 实例:简单小型操作系统 SLOS

我们开发了一个小操作系统,称为简单小型操作系统 SLOS(Simple Little Operating System)。它说明了 10.1 节讨论的基本模块是如何组成一个完整的操作系统的。这里选用 ARM 家族中最简单的核 ARM7TDMI,以 ADS1.2 为开发环境,以 ARM 公司的 Evaluator-7T 为目标板。移植 SLOS 到其它开发环境中也相对简单。这里使用第 10 章介绍的 Sandstone 固件来装载和执行 SLOS。

SLOS 是抢占式操作系统,周期性的中断唤醒睡眠态的(*dormant*)任务。为了简单起见,所有的任务和设备驱动程序都是静态的,即它们都在编译时创建而非系统运行时创建。

同时 SLOS 提供了一个设备驱动程序框架,这将在 11.2.7 小节中讨论。

SLOS 设计运行在无存储器管理单元或保护单元的 ARM7TDMI 核上。假设已在初始化代码(这里指第 10 章的 Sandstone)中配置好了存储器映射。要求将 SRAM 放在 $0x00000000 \sim 0x00080000$ 的地址内,将配置寄存器基地址放在 $0x03ff0000$ 。

SLOS 被装载到地址 $0x00000000$ (向量表就位于这里),这个地址也是 SLOS 的入口地址。当固件将控制权交出时,ARM 处理器工作在 SVC 模式。由于 SVC 模式是特权模式,所以允许初始化代码通过访问 cpsr 改变工作模式。可以利用这一点设置 IRQ 模式和系统模式下的堆栈。

在当前配置下,SLOS 包含 3 个任务和 2 个服务例程。任务 1 和任务 2 演示了使用一个二元信号量实现互斥的例子;实现的 2 个服务例程是周期性定时器(必需的)和按钮式中断(可选);任务 3 通过 ARM 评估板 Evaluator-7T 的串口提供一个简单的命令行接口。

SLOS 的每个任务都需要自己的堆栈。所有的任务都在用户模式下运行,因此任务只能读 cpsr 而不能写 cpsr。任务切换到特权模式的唯一途径是,使用一个 SWI 指令调用。这个机制也被用来调用设备驱动程序,因为设备驱动程序也可能需要写 cpsr。

在任务中可以修改 cpsr,但这种修改只能通过使用可更新 cpsr 条件标志的指令来间接进行。

11.2.1 SLOS 目录结构

SLOS 可以从原作者的网站下载,在第 11 章目录下。SLOS 的目录结构与 Sandstone 固件的目录结构(见图 10.1 和 11.1 节)相似。

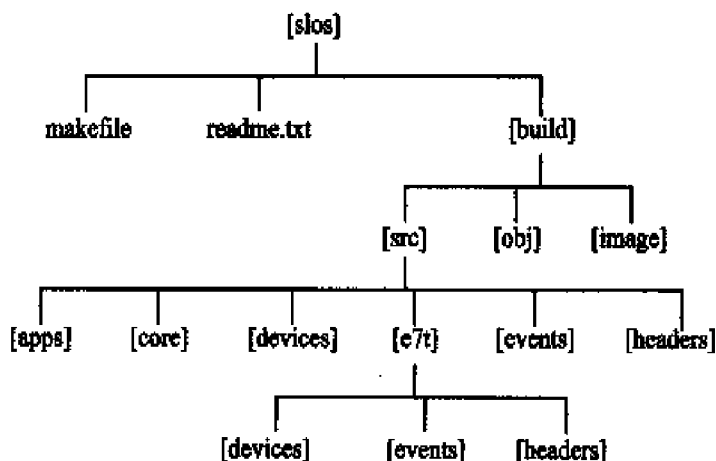


图 11.1 SLOS 目录结构

在包含操作系统所有源文件的目录 `slos/build/src` 下,有 6 个子目录。目录 `slos/build/src/core` 包含了各种工具的源文件,例如命令行解释器(CLI)的源代码。

特定平台的代码放在以此平台命名的目录下,例如,Evaluator - 7T 的特定代码就放在目录 `e7t` 下。

目录 `slos/build/src/devices` 存放所有的设备驱动程序源文件,目录 `slos/build/src/events` 存放处理服务、异常和中断的源文件。

最后,目录 `slos/build/src/apps` 存放特定配置下的应用程序(任务),例如,在 Evaluator - 7T 的实现上,有 3 个应用程序(任务)。

11.2.2 初始化

初始化 SLOS 有 3 个主要阶段——启动,建立进程控制块 PCB(Process Control Block)和执行 C 初始化代码。启动阶段设置 FIQ 寄存器和系统模式、SVC 模式、IRQ 模式下的堆栈。建立进程控制块阶段建立包含每个任务状态的 PCB,PCB 包括所有的 ARM 寄存器。在上下文切换时,PCB 用来保存和恢复任务状态,启动时 PCB 被设置成一个初始状态。最后的 C 初始化阶段调用设备驱动程序、事件处理程序和周期性定时器初始化例程。初始化一结束,就可以调用第一个任务了。

控制权通过复位向量转交给 SLOS。`vectorReset` 存放初始化代码的起始地址。假设固件将处理器置于 SVC 模式,这样就允许操作系统的初始化代码可完全访问 `cpsr`。第一条操作系统指令将初始化代码的起始地址(这里即 `coreInitialization`)装载到 `pc`。可以从下面列出的向量表中看到使用 `load` 指令装载一个字。汇编器使用 `pc` 与 `vectorReset` 地址的差来计算偏移量。

```
AREA ENTRY_SLOS, CODE, READONLY
```

```
ENTRY
```

```
LDR    pc,vectorReset
LDR    pc,vectorUndefined
LDR    pc,vectorSWI
LDR    pc,vectorPrefetchAbort
LDR    pc,vectorDataAbort
LDR    pc,vectorReserved
LDR    pc,vectorIRQ
LDR    pc,vectorFIQ
```

```
vectorReset      DCD    coreInitialize
```

ARM 嵌入式系统开发

vectorUndefined	DCD	coreUndefinedHandler
vectorSWI	DCD	coreSWIHandler
vectorPrefetchAbort	DCD	corePrefetchAbortHandler
vectorDataAbort	DCD	coreDataAbortHandler
vectorReserved	DCD	coreReservedHandler
vectorIRQ	DCD	coreIRQHandler
vectorFIQ	DCD	coreFIQHandler

作为初始化过程的一部分,这里通过使用备份(banked)FIQ 模式寄存器,实现了一个底层调试系统。这些寄存器用来保存状态信息。也并不总是使用 FIQ 寄存器,因为这些寄存器可能已用于其它目的了。

```
bringupInitFIQRegisters
```

```

MOV      r2,r14           ;保存 r14
BL       switchToFIQMode  ;切换到 FIQ 模式
MOV      r8,#0            ;r8_fiq = 0
MOV      r9,#0            ;r9_fiq = 0
MOV      r10,#0           ;r10_fiq = 0
BL       switchToSVCMode  ;切换到 SVC 模式
MOV      pc,r2            ;返回

```

```
coreInitialize
```

```
BL      bringupInitFIQRegisters
```

下一阶段是设置 SVC、IRQ 和系统堆栈基址寄存器。由于处理器已处于 SVC 模式,所以可直接设置 SVC 堆栈基址寄存器。代码如下:

```

MOV      sp,#0x80000       ;SVC stack
MSR      CPSR_c,#NoInt|SYS32md
MOV      sp,#0x40000       ;user/system stack
MSR      CPSR_c,#NoInt|IRQ32md
MOV      sp,#0x9000        ;IRQ stack
MSR      CPSR_c,#NoInt|SVC32md

```

如代码所列,建立堆栈以后,处理器将切换回 SVC 模式,以使初始化过程继续下去。在特权模式下,初始化的最后阶段就可通过清除 cpsr 的 I 位,并将处理器切换到用户模式,以允许 IRQ 中断。

执行完初始化启动代码后的结果如下：

- 底层调试机制被初始化；
- SVC、IRQ 和系统堆栈基址寄存器被设置。

要开始运行 SLOS,还必须初始化每个任务的 PCB。PCB 是一个保留的数据结构,用于保存所有 ARM 寄存器的一个副本(见表 11.1)。通过将相应任务的 PCB 数据复制到处理器寄存器,可激活某个任务。

表 11.1 任务控制块(PCB)

Offset	寄存器	Offset	寄存器
0	—	-36	r6
-4	r14	-40	r5
-8	r13	-44	r4
-12	r12	-48	r3
-16	r11	-52	r2
-20	r10	-56	r1
-24	r9	-60	r0
-28	r8	-64	pc+4
-32	r7	-68	spsr

每个任务的 PCB 必须在上下文切换发生前被初始化,因为上下文切换要将 PCB 数据复制到寄存器 r0~r15 和 cpsr 中。如果 PCB 未被初始化,则上下文切换将会复制一些不确定的数据到这些寄存器。

PCB 主要有 4 部分需要初始化:程序计数器、链接寄存器、用户模式堆栈和为每个任务保存的处理器状态寄存器(寄存器 r13,r14,r15 和 spsr)。

```

;void pcbSetUp(void *entryAddr, void *PCB, UINT offset);
pcbSetUp
    STR        r0,[r1,#-4]          ;PCB[-4]=C_TaskEntry
    STR        r0,[r1,#-64]         ;PCB[-64]=C_TaskEntry
    SUB        r0,sp,r2
    STR        r0,[r1,#-8]          ;PCB[-8]=sp-<offset>
    MOV        r0,#0x50             ;cpsr_c
    STR        r0,[r1,#-68]         ;PCB[-68]=iFt_User
    MOV        pc,lr

```

为便于说明 PCB 的初始化过程,这里摘取了初始化 PCB 例程的部分代码。例程 `pcb-Setup` 用于创建任务 2 和任务 3。其中寄存器 `r0` 是任务的入口地址——`entryAddr` 标号,也就是任务的执行地址;寄存器 `r1` 是 PCB 数据结构的地址——`pcbAddr` 标号,这个地址指向存放某个任务 PCB 的存储块;寄存器 `r2` 存放堆栈偏移值,用来定位堆栈在存储器映射中的位置。

注意:因为任务 1 是第一个执行的任务,所以它不需要初始化。

建立 PCB 的最后一步是设置当前任务的标识符,调度算法根据该标识符来决定当前应该运行哪个任务。

```
LDR    r0, = PCB_CurrentTask
MOV     r1, #0
STR     r1,[r0]
LDR     lr, = C_Entry
MOV     pc,lr           ,enter the CEntry world
```

在程序段的最后调用了第一个 C 例程——`C_Entry`,这通过将此例程的起始地址赋给 `pc` 来实现。

PCB 初始化过程执行完毕的结果如下:

- 初始化了所有 3 个任务的 PCB;
- 设置当前要执行的 PCB 为任务 1(标识符为 0)。

现在初始化过程交给例程 `C_Entry`,该例程在 `build/src/core/cinit.c` 源文件里。`C_Entry` 例程调用另一个例程 `cinit_init()`,`cinit_init()` 例程(代码在下面给出)初始化设备驱动程序、事件服务和周期性中断。这个 C 程序设计成不需要标准 C 库的支持,因为它没调用任何标准 C 库中的函数,例如 `printf()` 和 `fopen()` 等。

```
void cinit_init(void)
{
    eventIODeviceInit();
    eventServicesInit();
    eventTickInit(2);
}
```

函数 `eventIODeviceInit()`,`eventServicesInit()` 和 `eventTickInit()` 分别用来初始化操作系统的各个不同部分。`eventTickInit()` 只有一个参数,值为 2。这个参数用来设置周期性中断的时间间隔(以 ms 为单位)。

cinit_init()例程的初始化完成以后,周期性定时器就被启动了(代码参见下面的 C_Entry 例程)。这就意味着任务 1 应该在定时器第一次中断之前被调用。为使周期性事件可以中断处理器,必须使能 IRQ,且处理器必须处于用户模式下。将这些设置完成后,就可以调用任务 1 的入口地址 C_EntryTask1。

```
int C_Entry(void)
{
    cinit_init();
    eventTickStart();
    __asm
    {
        MSR        CPSR_c, # 0x50
    }

    C_EntryTask1();
    return 0;
}
```

如果程序正常运行,那么 C_Entry 例程最后的 return 语句将永远不会执行到。至此,完成了所有初始化工作,操作系统就可正常工作了。

所有 C 初始化代码执行完毕的结果如下:

- 设备驱动程序被初始化;
- 事件服务被初始化;
- 周期性定时器被初始化并启动;
- 在 cpsr 中使能了 IRQ 中断;
- 处理器工作在用户模式;
- 调用了任务 1 的入口地址(C_EntryTask1)。

11.2.3 存储模型

SLOS 采用一个简单的存储模型,如图 11.2 所示。SLOS 的代码部分(包括任务)分配在存储器低端,IRQ 和每个任务的堆栈分配在存储器高端,SVC 堆栈分配在存储器顶端。图 11.2 中的箭头表示堆栈的增长方向。

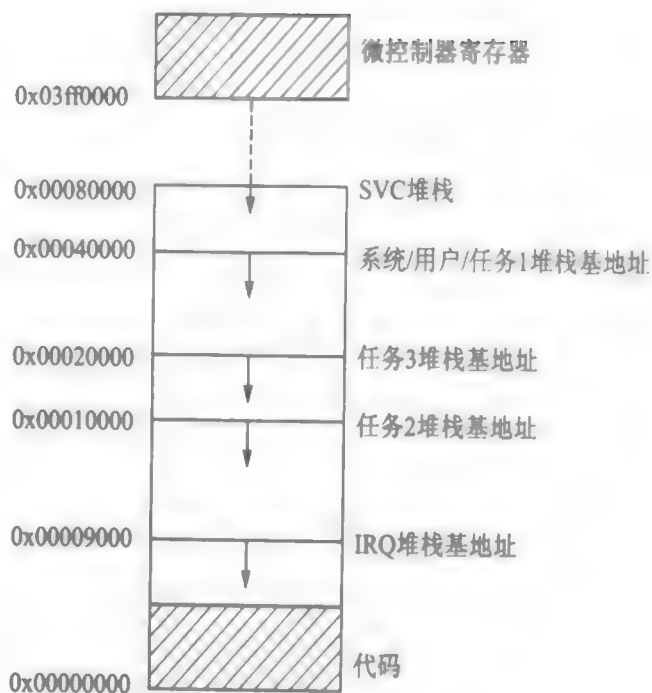


图 11.2 存储器映射

11.2.4 中断和异常处理

SLOS 操作系统的实现实际上只用到 3 个异常，其它异常都被忽略。当遇到其它未使用

表 11.2 异常分配

异常	目的
复位	初始化操作系统
SWI	设备驱动程序访问机制
IRQ	事件服务机制

的异常时，便进入相应的哑处理程序。为安全起见，这里将这些哑处理程序设计成死循环。为完善这个操作系统，这些哑处理程序需要用完整的处理程序来替换。3 个异常及其在操作系统中的使用方法如表 11.2 所列。

11.2.4.1 复位异常

复位向量只在初始化阶段被调用一次。理论上，可以再次调用它来重新初始化系统——例如，由看门狗定时器溢出引发的处理器复位。当系统长时间没反应时，看门狗定时器可以用来复位系统。

11.2.4.2 SWI 异常

应用程序通过 SWI 处理机制来调用设备驱动程序。SWI 指令迫使处理器从用户模式切换到 SVC 模式。内核的 SWI 处理程序在下面列出，处理程序的第一步是将寄存器 r1~r12 保

存到 SVC 堆栈中。

下一步是计算 SWI 指令的地址,并将指令装载到寄存器 r10。将 SWI 指令的高 8 位屏蔽后,便得到了 SWI 号;然后将 SVC 堆栈的地址复制到寄存器 r1,并以其作为调用 SWI C 处理程序的第 2 个参数。

然后将 spsr 复制到寄存器 r2,并保存到堆栈中。这一步只有当发生 SWI 嵌套调用时才需要。接下来处理程序将跳转到调用 C 处理例程的代码处。

```
coreSWIHandler
    STMFD    sp!,{r0-r12,r14}      ;保存上下文
    LDR      r10,[r14,#-4]          ;装载 SWI 指令
    BIC      r10,r10,#0xff000000    ;屏蔽高 8 位
    MOV      r1,r13                 ;将 r13_svc 复制到 r1
    MRS      r2,spsr                ;将 spsr 复制到 r2
    STMFD    r13!,{r2}              ;将 r2 保存到堆栈
    BL       swi_jumtable            ;跳转到 swi_jumtable
```

使用 BL 指令后面的代码返回到调用 SWI 的程序,如下面代码所示,从堆栈中恢复 spsr 和所有用户模式下的寄存器(包括 pc)。

```
LDMFD      r13!,{r2}                ;恢复 r2(spsr)
MSR        spsr_cxsf,r2              ;将 r2 复制到 spsr
LDMFD      r13!,{r0-r12,pc}^         ;恢复上下文并返回
```

BL 指令设置链接寄存器。当 SWI C 处理程序完成时,将执行以下代码。

```
swi_jumtable
    MOV      r0,r10                  ;将 SWI 号复制到 r0
    B        eventsSWIHandler        ;跳转到 SWI 处理程序
```

C 处理程序 eventsSWIHandler(如下代码所列)被调用,并且寄存器 r0 包含 SWI 号,寄存器 r1 指向保存在 SVC 堆栈中的寄存器。

```
void eventsSWIHandler(int swi_number, SwiRegs * r)
{
    if (swi_number == SLOS)
    {
        if (r->r[0] == Event_IODeviceInit)
        {
            /* 不使能 IRQ 中断 ... */
            ioInitializeDrivers();
        }
    }
}
```

```
else
{
    /* 若非初始化,则切换到系统模式,并使能 IRQ */
    if (STATE!=1) {modifyControlCPSR (SYSTEM|IRQoN);}

    switch (r->r[0])
    {
        case /* SWI */ Event_IODeviceOpen:
            r->r[0] =
                (unsigned int) io_open_driver
                (
                    /* int * ID */ (UID *)r->r[1],
                    /* unsigned major_device */ r->r[2],
                    /* unsigned minor_device */ r->r[3]
                );
            break;
        case /* SWI */ Event_IODeviceClose:
            /* 调用 io_close_driver */
            break;
        case /* SWI */ Event_IODeviceWriteByte:
            /* 调用 io_writebyte_driver */
            break;
        case /* SWI */ Event_IODeviceReadByte:
            /* 调用 io_readbyte_driver */
            break;
        case /* SWI */ Event_IODeviceWriteBit:
            /* 调用 io_writebit_driver */
            break;
        case /* SWI */ Event_IODeviceReadBit:
            /* 调用 io_readbit_driver */
            break;
        case /* SWI */ Event_IODeviceWriteBlock:
            /* 调用 io_writeblock_driver */
            break;
        case /* SWI */ Event_IODeviceReadBlock:
            /* 调用 io_readblock_driver */
            break;
```

```

}
/* 若非初始化,切换回管理模式并禁用 IRQ */
if (STATE!= 1) {modifyControlCPSR (SVC|IRQoFF);}
}
}
}

```

11.2.4.3 IRQ 异常

IRQ 处理程序比 SWI 处理程序简单得多,将它设计为不支持嵌套的基本中断处理程序。处理程序首先保存上下文;然后将中断控制器的中断请求寄存器 INTPND 复制到寄存器 r0;接下来每个中断服务例程将其自身所代表的中断源与寄存器 r0 进行比较,如果匹配,则该例程被调用,否则将寄存器 INTPND 所表示的中断看作虚中断并忽略它。

```

TICKINT      EQU      0x400
BUTTONINT    EQU      0x001

```

eventsIRQHandler

```

SUB          r14, r14, #4           ;r14_irq -= 4
STMFD        r13!, {r0-r3, r12, r14} ;保存上下文
LDR          r0, INTPND             ;r0 = 中断状态寄存器
LDR          r0, [r0]               ;r0 = memory[r0]
TST          r0, #TICKINT           ;若定时器中断
BNE          eventTickVeneer        ;则定时器中断服务程序
TST          r0, #BUTTONINT         ;若按键中断
BNE          eventButtonVeneer      ;则按键中断服务程序
LDMFD        r13!, {r0-r3, r12, pc}~ ;返回到任务

```

对于系统认可的中断源,将调用各自的中断服务例程(veneer)来处理这个中断事件。下面的代码是一个定时器中断服务例程,从例子中可看出该中断服务例程包含 2 个调用:第 1 个是复位定时器例程 eventsTickService(平台特有的调用);第 2 个是称为调度程序的 kernelScheduler 例程,该例程依次调用上下文切换。

eventTickVeneer

```

BL          eventTickService        ;复位定时器硬件
B           kernelScheduler         ;跳转到调度程序

```

不需要将寄存器 r4~r12 放到 IRQ 堆栈中,因为调度算法和上下文切换将会仔细处理这些寄存器。

11.2.5 调度程序

SLOS 的底层调度程序(或称为分发程序,dispatcher)采用简单的静态循环算法,如下面的伪代码所示。这里的“静态”是指,任务只能在操作系统初始化时创建。SLOS 中的任务在操作系统处于活动状态时,既不能创建,也不能删除。

```
task t = 0, t';

scheduler()
{
    t' = t + 1;
    if t' = MAX_NUMBER_OF_TASKS then
        t' = 0 // 第一个任务
    end;

    ContextSwitch(t, t')
}
```

如前面所述,在初始化阶段,将当前活动任务 t 的 PCB 地址 PCB_CurrentTask 置为任务 0。当定时器中断产生时,新任务 t' 等于当前任务号 t 加 1。如果任务号等于任务数最大值 MAX_NUMBER_OF_TASKS,则将任务 t' 重新置为任务 0。

表 11.3 列出了调度程序所使用的标号及其在算法中的含义。下面关于调度程序的过程和代码描述都会用到这些标号。

表 11.3 调度程序使用的变量

变 量	描 述
PCB_CurrentTask	保存当前任务 t
PCB_Table	每个任务 PCB 的地址指针表
PCB_PtrCurrentTask	当前任务 t 的指针
PCB_PtrNextTask	下一个任务 t' 的指针
PCB_IRQStack	IRQ 堆栈的临时存储空间(上下文切换)

- ① 从 PCB_CurrentTask 中获得当前任务 ID;
- ② 在 PCB_Table 中使用 PCB_CurrentTask 作为索引找到当前任务对应的 PCB 地址;
- ③ 使用步骤②所获得的地址来更新 PCB_PtrCurrentTask 的值;

- ④ 使用循环算法计算新任务 t' 的 ID;
- ⑤ 将新任务 t' 的 ID 保存到 PCB_CurrentTask;
- ⑥ 在 PCB_Table 中使用更新后的 PCB_CurrentTask 作为索引找到下一个任务的 PCB 地址;
- ⑦ 将下一个任务的 PCB 地址保存到 PCB_PtrNextTask。

调度下一个任务 t' 的代码如下:

```

MaxNumTasks    EQU    3
FirstTask      EQU    0

CurrentTask

    LDR    r3, = PCB_CurrentTask          ;[1] r3 = PCB_CurrentTask
    LDR    r0,[r3]                        ;r0 = 当前任务 ID
    LDR    r1, = PCB_Table                ;[2] r1 = PCB_Table address
    LDR    r1,[r1,r0,LSL#2]               ;r1 = mem32[r1 + r0 << 2]
    LDR    r2, = PCB_PtrCurrentTask       ;[3] r2 = PCB_PtrCurrentTask
    STR    r1,[r2]                        ;mem32[r2] = r1 ; 任务地址
; * * PCB_PtrCurrentTask - updated with 当前任务的地址
; * * r2 = PCB_PtrCurrentTask 地址
; * * r1 = 当前任务 PCB 地址
; * * r0 = 当前任务 ID

NextTask

    ADD    r0,r0,#1                       ;[4] r0 = (CurrentTaskID) + 1
    CMP    r0,#MaxNumTasks                ;if r0 == MaxNumTasks
    MOVEQ  r0,#FirstTask                  ;then r0 = FirstTask (0)
    STR    r0,[r3]                        ;[5] mem32[r3] = next Task ID
    LDR    r1, = PCB_Table                ;[6] r1 = PCB_Table addr
    LDR    r1,[r1,r0,LSL#2]               ;r1 = memory[r1 + r0 << 2]
    LDR    r0, = PCB_PtrNextTask          ;[7] r0 = PCB_PtrNextTask
    STR    r1,[r0]                        ;memory[r0] = next task addr

```

执行完调度程序以后的结果如下:

- PCB_PtrCurrentTask 指向当前活动的 PCB 地址;
- PCB_PtrNextTask 指向下一个活动的 PCB 地址;
- PCB_CurrentTask 保存下一个任务的 ID 值。

11.2.6 上下文切换

上下文切换使用调度程序产生的更新信息,交换活动任务 t 和下一个任务 t' 。为达到这一目的,上下文切换将这个过程分成 2 个阶段,如图 11.3 所示。第一阶段将处理器寄存器保存到由 PCB_PtrCurrentTask 指向的当前任务 t 的 PCB 中;第二阶段将由 PCB_PtrNextTask 指向的下一任务 t' 的 PCB 数据恢复到寄存器中。

下面简单介绍上下文切换 2 个阶段的过程和代码,先是保存当前任务的上下文,然后恢复新任务的上下文。

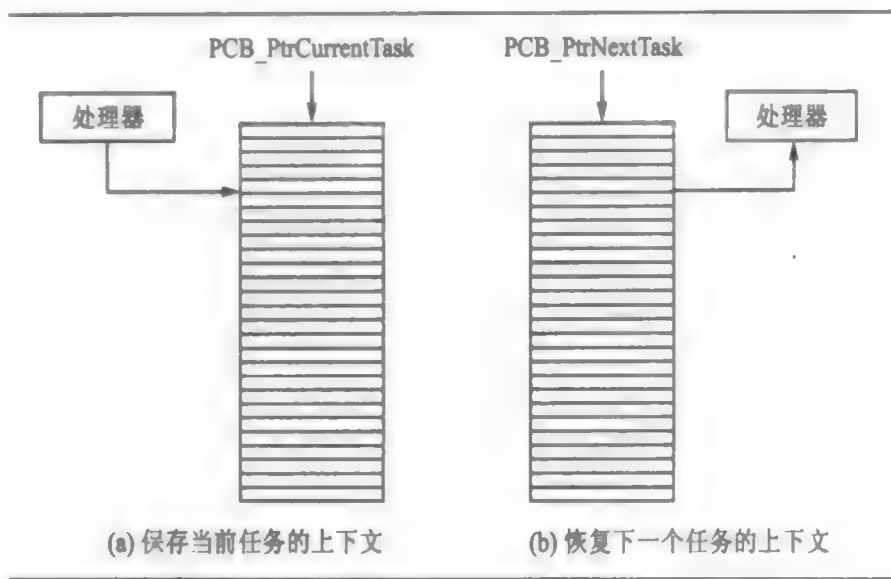


图 11.3 上下文切换

11.2.6.1 保存当前任务上下文

第一阶段是保存当前活动任务 t 的寄存器。所有的任务都在用户模式下运行,所以必须保存用户模式下的寄存器。步骤如下:

- ① 必须从堆栈中恢复寄存器 $r0 \sim r3$ 和 $r14$,这些寄存器属于当前任务;
- ② 寄存器 $r13$ 用于指向当前任务 PCB_CurrentTask 的 PCB,偏移量为 -60 ,这个偏移量允许 2 条指令来更新整个 PCB;
- ③ 保存所有用户模式下的寄存器 $r0 \sim r14$,这只要用一条指令来完成。符号“ \sim ”表示对用户模式下寄存器的多次存储操作。第二条存储指令保存 $spsr$ 和用于返回的链接寄存器。

将寄存器保存到 PCB 中的代码为

Offset15Regs EQU 15 * 4

handler_contextswitch

LDMFD	r13!, {r0 - r3, r12, r14}	;[1.1] 恢复寄存器
LDR	r13, = PCB_PtrCurrentTask	;[1.2]
LDR	r13, [r13]	; r13 = mem32[r13]
SUB	r13, r13, # Offset15Regs	; r13 -= 15 * Reg, place r13
STMIA	r13, {r0 - r14}^	;[1.3] 保存用户模式下的寄存器
MRS	r0, spsr	; 复制 spsr
STMDB	r13, {r0, r14}	; 保存 r0(spsr) 和 r14(lr)

保存当前任务上下文以后的结果如下:

- 复位 IRQ 堆栈并将它保存到 PCB_IRQTask;
- 任务 t 的用户模式下的寄存器被保存到当前 PCB 中。

11.2.6.2 恢复新任务上下文

上下文切换的第二阶段是将 t' 的 PCB 恢复到用户模式下的寄存器中。完成了这一过程, 例程接着就必须将控制权转交给新任务 t' 。步骤如下:

- ① 恢复并将 r13 设置到与新任务 PCB 的起始地址偏移为 -60 的位置;
- ② 首先恢复寄存器 spsr 和链接寄存器, 然后恢复下一个任务的寄存器 r0~r14, 寄存器 r14 是用户模式下的寄存器 r14, 而不是指令中带有符号“~”的 IRQ 寄存器 r14;
- ③ 从 PCB_IRQStack 恢复 IRQ 堆栈;
- ④ 复制保存在寄存器 r14 中的地址到 pc, 并更新 cpsr, 以继续运行新任务。

从 PCB 中恢复寄存器的代码如下:

LDR	r13, = PCB_PtrNextTask	;[2.1] r13 = PCB_PtrNextTask
LDR	r13, [r13]	; r13 = mem32[r13]; 下一个任务 PCB
SUB	r13, r13, # Offset15Regs	; r13 -= 15 * Registers
LDMDB	r13, {r0, r14}	;[2.2] 装载 r0 和 r14
MSR	spsr_cxsf, r0	; spsr = r0
LDMIA	r13, {r0 - r14}^	; 装载 r0_user - r14_user
LDR	r13, = PCB_IRQStack	;[2.3] r13 = IRQ 堆栈地址
LDR	r13, [r13]	; r13 = mem32[r13]; 复位 IRQ
MOVS	pc, r14	;[2.4] 返回到下一个任务

恢复新任务上下文以后的结果如下:

- 上下文切换完成;

- 新任务的寄存器恢复到用户模式下的寄存器中；
- IRQ 堆栈恢复到进入 IRQ 中断处理程序之前的设置。

11.2.7 设备驱动程序框架

使用 SWI 指令实现设备驱动程序框架 *DDF(Device Driver Framework)*。DDF 保护操作系统,使应用程序不能直接访问硬件,并提供一个统一的标准接口给所有任务。如果任务想访问一个特定设备,则它必须先获得一个惟一的标识号 UID。这通过调用打开宏(*eventIODeviceOpen*)来完成。这个宏被直接转换成一条设备驱动程序 SWI 指令。UID 用来保证没有其它任务正在访问相同的设备。

打开一个设备驱动程序的任务代码如下:

```
device_treestr * host;
UID serial;

host = eventIODeviceOpen(&serial, DEVICE_SERIAL_E7T, COM1);

if (host == 0)
{
    /* ... error device driver not found ... */
}

switch (serial)
{
case DEVICE_IN_USE;
case DEVICE_UNKNOWN;
/* ... problem with device ... */
}
```

这个例子说明了如何使用设备驱动程序框架来打开一个串行设备。

可以使用一些宏将参数传送到寄存器 *r1~r3*,然后这些寄存器通过 SWI 机制传给设备驱动函数。在这个例子中,实际上只有寄存器 *r1* 指向的值(&serial)是被更新的。这个值用来返回 UID,如果返回值为 0,则表示出错。

下面的代码显示了宏 *eventIODeviceOpen* 如何转化成一条 SWI 指令调用:

```
PRE    r0 = Event_IIODeviceOpen (unsigned int)
        r1 = &serial (UID * u)
        r2 = DEVICE_SERIAL_E7T (unsigned int major)
```

```
r3 = COM1 (unsigned int minor)
```

```
SWI 5075
```

POST r1 = UID 指针指向的数据改变了

当任务运行在非特权模式下时,可使用 SWI 切换到特权模式,这样就允许设备驱动程序完全访问 cpsr。图 11.4 说明了当调用一个设备驱动程序函数时实际的模式变化。从图中可看出设备驱动程序自己在系统模式(为特权模式)下执行。

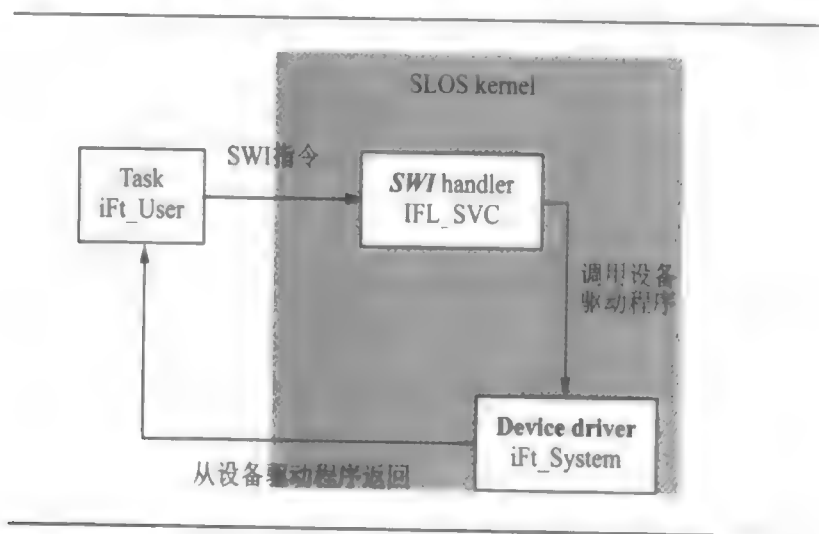


图 11.4 调用设备驱动程序

一旦执行 SWI 指令,处理器就进入 SVC 模式,并且 IRQ 中断自动关闭。只有当处理器切换到系统模式时,中断才能继续使用。只有一个例外,那就是在初始化阶段调用设备驱动程序。在这种情况下,中断仍然是禁止的。

11.3 总 结

一个运行在 ARM 处理器上的嵌入式操作系统的基本模块如下:

- 初始化模块(initialization) 建立所有的内部变量、数据结构和操作系统使用的硬件设备。
- 存储器处理模块(memory handling) 组织存储器,以容纳内核和要执行的各种应用程序。
- 所有的中断和异常都需要一个处理程序,对于未使用的中断和异常必须提供一个哑处理程序。

ARM 嵌入式系统开发

- 抢占式操作系统需要一个周期性定时器,该定时器产生周期性的中断,以调用调度程序。
- 调度程序是用来决定下一个要执行哪个任务的算法。
- 上下文切换保存当前任务的状态并恢复下一个任务的状态。

以下这些模块在一个被称为简单小型操作系统(SLOS)的操作系统里都有相应的例子:

- 初始化模块——建立 SLOS 的所有功能模块,包括模式堆栈、每个应用程序的进程控制块(PCB)和设备驱动程序等。
- 存储模型——SLOS 内核位于存储器低地址,每个应用程序都有自己的存储空间和堆栈,微控制器系统寄存器独立于 ROM 和 SRAM。
- 中断和异常——SLOS 只使用 3 个事件:复位、SWI 和 IRQ,所有的未使用的中断和异常都提供一个哑处理程序(如死循环)。
- 调度程序——SLOS 实现一个简单的循环调度程序。
- 上下文切换——首先将当前上下文保存到一个 PCB 中,然后从另一个 PCB 中恢复下一个任务的上下文。
- 设备驱动程序框架——保护操作系统,以使应用程序不能直接访问硬件。

第 12 章

高速缓冲存储器 cache

- 存储层次和 cache
- cache 结构
- cache 策略
- 协处理器 15 与 cache
- 清除和清理 cache
- cache 锁定
- cache 与软件性能
- 总 结

ARM 嵌入式系统开发

cache 是一种容量小,速度快的存储器阵列。它位于主存和处理器内核之间,保存着最近一段时间处理器涉及到的主存块内容。在需要进行数据读取操作时,为了改善系统性能,处理器尽可能从 cache 中读取数据,而不是从主存中获取数据。cache 的主要目标就是,减小慢速存储器给处理器内核造成的存储器访问瓶颈问题的影响。

cache 经常与写缓冲器(*write buffer*)一起使用。写缓冲器是一个非常小的先进先出(FIFO)存储器,位于处理器核与主存之间。使用写缓冲器的目的是,将处理器核和 cache 从较慢的主存写操作中解脱出来。*

cache 是一个法语单词,意思是“隐蔽的存储场所”。将 cache 使用到 ARM 嵌入式系统中,这个定义就显得更加贴切了。cache 存储器和写缓冲器加到处理器内核上之后,对软件代码的执行是透明的。这样在一个拥有 cache 的处理器内核上运行以前写的软件时,代码就不需要重新编写。虽然 cache 和写缓冲器都有附加的控制硬件,可以自动处理主存与处理器之间的代码和数据的传送,但是了解处理器 cache 的设计细节,可以帮助编程人员在特定的 ARM 核上编写出执行更快的程序。

本章主要描述 cache 能做的许多有效的工作,以使程序执行得更快。但是在系统中使用 cache 是否会带来其它弊端呢?答案是肯定的。其中最主要的弊端就是,很难判断一个程序的执行时间。下面将对此作出解释。

因为 cache 存储器只提供了主存中非常少的一部分数据,在程序执行过程中,cache 会很快被填满。一旦被填满,cache 控制器就会频繁地从 cache 存储器中移出原来的代码和数据,以给新的代码和数据留出存储空间。这种移出操作一般是随机发生的,它会留下一部分数据而将其它部分移出。这样,在任何一个给定的时刻,某个数值可能在 cache 中,也可能不在。

既然数据在任何一个给定的时刻可能在 cache 中,也可能在主存中,由于直接使用 cache 中的数据和从主存中装载 cache 的一行数据所需时间不一样,因此一个程序每次执行所需时间就会有轻微的差别。

本章将介绍 cache 在标准的存储层次中所处的位置,以及存储器访问的局部性原理,以此来解释 cache 是如何改善系统性能的;然后简单介绍一下 cache 的体系结构,并定义一系列 ARM 常用术语;最后,提供一些示例代码,示范如何清理(*clean*)和清除(*flush*) cache,以及如何将代码段和数据段锁定在 cache 中。

* 写缓冲器与 cache 在存储层次上处于同一层次,只是在进行主存写操作时才使用写缓冲器。——译者注

12.1 存储层次和 cache

第1章介绍了计算机系统存储器的分层结构,图12.1更好地显示了 cache 和写缓冲器在存储器层次结构中的位置。

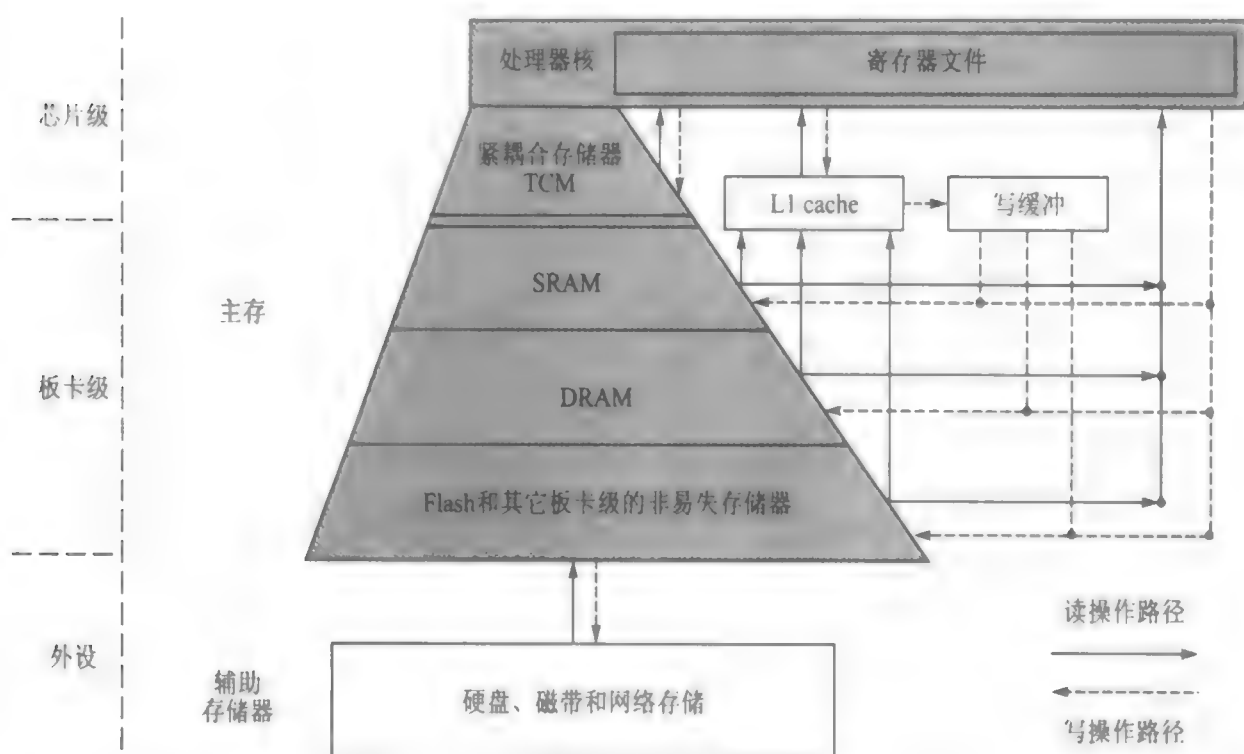


图 12.1 存储器的层次结构

存储层次的最内(顶)层在处理器内核中。该存储器与处理器的结合非常紧密,以致在很多情况下,很难将两者分开。该存储器被称为寄存器文件(register file)。这些寄存器被集成在处理器内核中,在系统中提供最快的存储访问。

接下来是一级存储器,存储部件与处理器内核通过专用的片上接口连接起来。紧耦合存储器(TCM)和一级 cache 就在这一层。后面会介绍更多关于 cache 的内容。

主存也在第一级里,包含一些易失性(即掉电后会导致数据丢失)存储器,如 SRAM、DRAM 以及一些非易失性存储器(掉电后数据不会丢失)如 Flash 存储器。主存的主要任务就是承载在系统中运行着的程序。

再下一层是二级存储器(辅助存储器),即低速的相对较便宜的大容量存储设备,如硬盘存储器和可移动存储器。在这一层里,还有从外围设备得到的数据,其访问时间特别长。

ARM 嵌入式系统开发

二级存储器用来存储正在运行的较大程序的未被使用的部分(由于程序过大,不适合将全部程序都放在主存中),或者存放当前没有运行的程序。

需要指出的是,存储层次结构不但依赖于体系结构的设计,也依赖于相关的工艺和技术。例如,TCM 和 SRAM 在技术上相同,但在结构排列上不同:TCM 在片上,而 SRAM 在板上。

在存储层次中,cache 可以被放置于存在明显访问速度差异的任何层次上,并且可以改善系统性能。cache 存储器系统把存储层次中较低层次的信息取出,并把它们临时存放到较高的层次中。

图 12.1 包含了一个 L1 cache 和一个写缓冲器。L1 cache 是一个高速片上存储阵列,用来临时承载低层存储器中的程序和代码。cache 所装载的信息可以缩短访问指令和数据所需要的时间。写缓冲器则是一个容量很小的 FIFO 缓冲器,其主要作用就是对由 cache 中写到主存的数据提供缓冲。

图 12.1 中没有把 L2 cache 标记出来,它应该位于 L1 cache 和更低层的存储器之间。通常也把 L1 cache 和 L2 cache 分别称作一级 cache 和二级 cache。

从图 12.2 中可以看出 cache 与主存储器系统和处理器内核之间的关系。图的上半部分是一个没有 cache 的系统,处理器内核以自己支持的数据类型方式直接访问主存。图的下半部分是一个带有 cache 的系统,cache 的存取速度比主存快得多,这样处理器内核对数据的访问请求可有快速的响应。cache 与主存的关系主要体现在高速 cache 和低速主存之

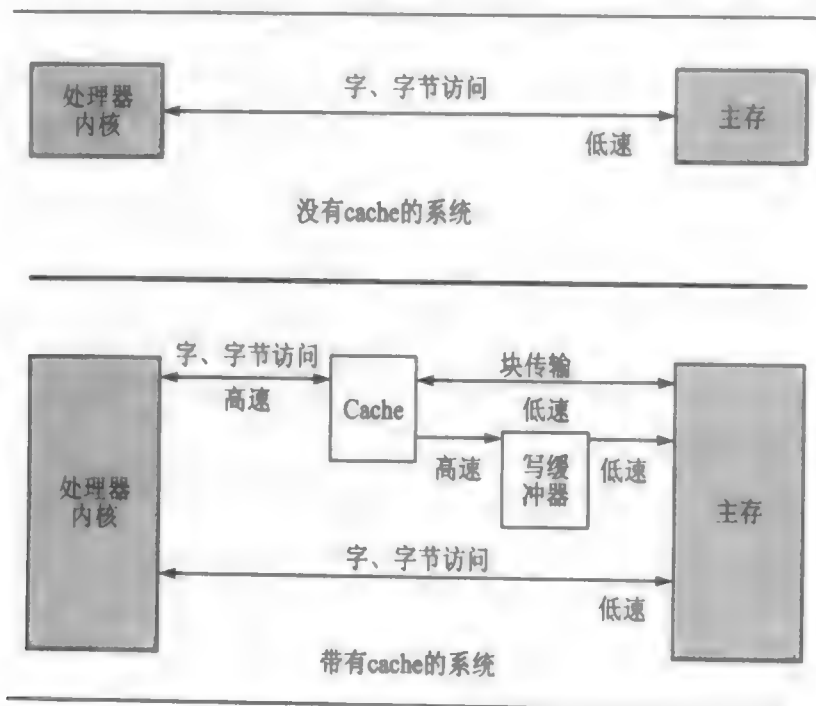


图 12.2 cache、处理器内核及主存之间的关系

间传送小块数据上,这样的小块数据被称作 cache 行。写缓冲器作为临时缓冲器帮助 cache 释放存储空间,即从 cache 中搬出的数据暂存在写缓冲器中。cache 控制器将 cache 行以较高的速度放到写缓冲器中,之后写缓冲器以较低速度将该 cache 行写入主存中。

cache 和存储器管理单元

如果带 cache 的处理器内核支持虚拟存储,那么 cache 就可以被放在处理器内核与存储器管理单元 MMU 之间,或者在 MMU 与物理存储器(主存)之间。cache 放置在 MMU 之前或之后,决定了 cache 的寻址范围和编程结构(程序员看到的 cache 结构)。图 12.3 显示了 cache 在系统中的不同位置带来的差异。

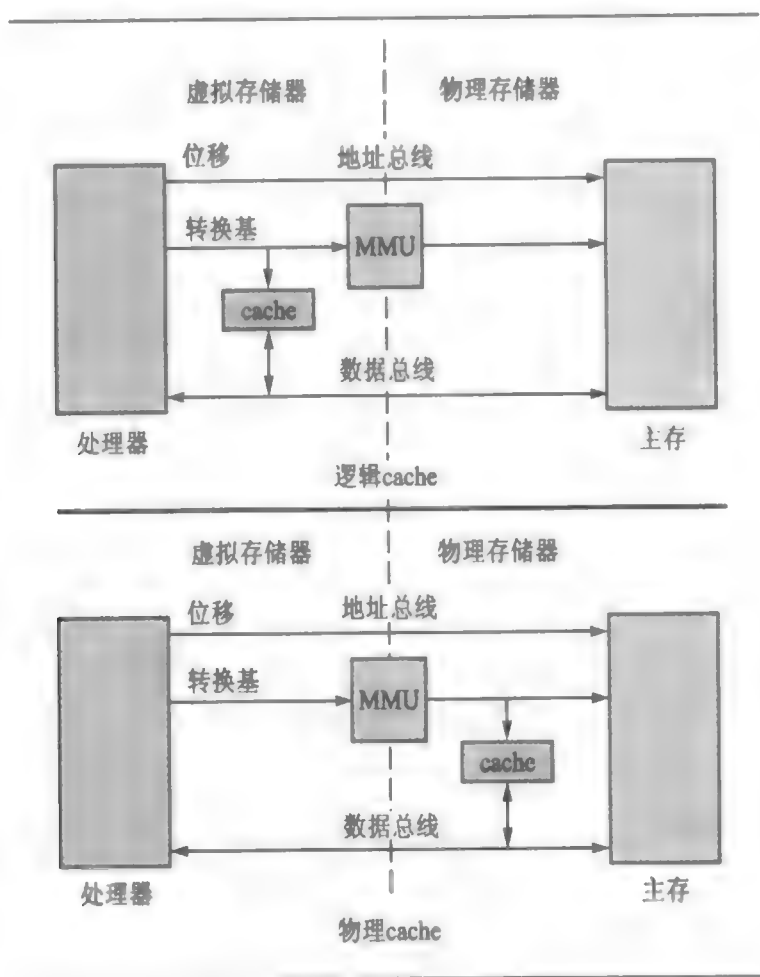


图 12.3 逻辑 cache 和物理 cache

逻辑 cache 在虚拟地址空间存储数据,它位于处理器和 MMU 之间。处理器可以直接通过逻辑 cache 访问数据,而无须通过 MMU。逻辑 cache 又被称作**虚拟 cache**。

物理 cache 使用物理地址存储数据,它位于 MMU 和主存之间。当处理器访问存储器时,MMU 必须先把虚拟地址转换成物理地址,cache 存储器才可以向内核提供数据。

带有 cache 和 MMU 的 ARM 处理器中,从 ARM7 到 ARM10,包括 Intel StrongARM 和 Intel XScale 处理器,都使用逻辑 cache。ARM11 处理器(ARMv6 体系结构)系列使用物理 cache。有关 MMU 的其它内容可以参考本书第 14 章。

使用 cache 来改进性能是可行的,因为计算机程序的执行并不是随机的。程序执行的可预测性是 cache 系统成功的关键。如果程序对存储器的访问是随机的,那么 cache 对整个系统的综合性能几乎不会有什么改进。*存储器访问的局部性原理*可以很好地解释为什么在系统中加入 cache 可以改善性能。这个原理表明:程序在执行过程中会频繁地运行小范围的循环代码,而这些循环又会对数据存储器中的局部区域反复访问。

对存储器中相同或邻近的数据和代码反复使用,是 cache 改善性能的主要原因。处理器在第一次访问存储器时,将相关数据和程序加载到 cache 中,使随后的访问速度大大提高。对高速 cache 的重复访问改善了系统的性能。

cache 同时使用了时间和空间的局部性原理。如果对存储器的访问受时间影响,在时间上有连续性,则这种时间上密集访问被称为时间局部性访问;如果多次对存储器访问的地址相近,则这种空间上邻近的访问被称作空间局部性访问。

12.2 cache 结构

带有 cache 的 ARM 内核采用了两种总线结构:冯·诺依曼结构和哈佛结构。这两种总线结构的区别在于,是否在内核与主存之间将指令和数据通道分离。分别有不同的 cache 设计来支持这两种结构。

在使用冯·诺依曼结构的处理器内核中,只有一个数据和指令公用的 cache。这种类型的 cache 被称作统一 cache(或混合 cache),它可以存储指令和数据。

哈佛结构将指令总线 and 数据总线分离,以改善系统的综合性能,但是支持两种总线需要两种 cache。所以在使用哈佛结构的处理器核中,存在两种 cache:指令 cache(I-cache)和数据 cache(D-cache)。这种类型的 cache 被称作分离 cache(split cache)。在分离 cache 中,指令被存储在指令 cache 中,而数值被存储在数据 cache 中。

可以通过图 12.4 中的统一 cache 来了解 cache 的基本结构。cache 的两个主要组成部分是 cache 控制器和 cache 存储器。cache 存储器是一个专用的存储器阵列,其访问单元称为 cache 行。cache 控制器使用处理器在访问存储器时所提供的地址的不同段,以选择 cache 存储器的不同部分。下面将首先介绍 cache 存储器的结构,接下来介绍 cache 控制器的一些细节。

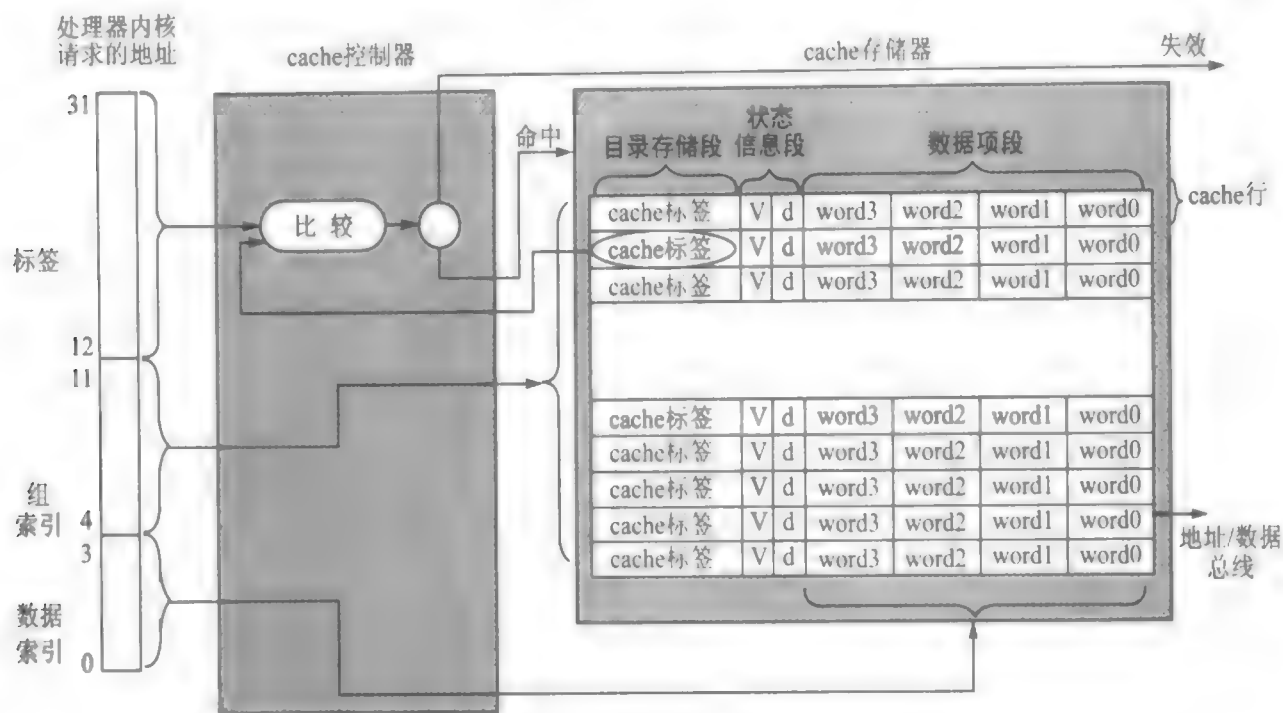


图 12.4 拥有 256 行的 4 KB cache, 每行 4 个 32 位字(word)

12.2.1 cache 存储器的基本结构

图 12.4 的右侧是一个简单的 cache 存储器。它有 3 个主要部分:目录存储段(directory store)、状态信息段(status information)和数据项段(data section)。每一个 cache 行都由这 3 部分来表示。

cache 必须知道 cache 存储器中的每个 cache 行所对应于主存中的位置,cache 使用目录存储段来记录每个 cache 行是由主存的什么地方拷贝而来。该目录项被称作“cache 标签”(cache-tag)。

同样,cache 存储器必须存储来自主存的信息,这些信息被放在数据项段里(见图 12.4)。

cache 的大小是由 cache 可以存储的主存中实际数据和代码的大小决定的。在计算 cache 容量时,用来存储 cache 标签和状态信息位的那部分 cache 存储器不计算在内。

在 cache 存储器中,还有用来记录状态信息的状态位。2 个常见的状态位是有效位(valid bit)和脏位(dirty bit)。有效位用来标记当前的 cache 行是活动的,即该 cache 行中包含最初从主存中取得的数据,并可以为处理器内核所用。脏位则用来标记该 cache 行中所含的内容与主存中相应的内容是否一致。在 12.3.1 小节中,将会详细地解释脏位的含义。

12.2.2 cache 控制器的基本操作

cache 控制器是一种硬件,它将主存中的数据或者代码自动拷贝到 cache 存储器中。cache 控制器在不为应用软件所知的情况下,自动完成搬移工作。所以,同一个应用软件不用修改,就可以在有 cache 和没有 cache 的系统中运行。

读/写存储器的请求在被传送到存储器控制器之前,会被 cache 控制器截获,cache 控制器将该请求的地址信息分成 3 部分:标签域(tag field)、组索引域(set index field)和数据索引域(data index field)。3 个位域分别见图 12.4。

首先,控制器通过组索引域在 cache 存储器中确定可能包含所要求的代码和数据的 cache 行的位置,即确定某一 cache 行。cache 行中还包含 cache 标签和状态位,控制器就是通过它们来确定数据的实际存储位置的。

接下来,控制器检查有效位,以确定该 cache 行当前是否处于活动状态,并且将请求地址的标签域的值与 cache 标签比较。如果 cache 行当前是活动的,并且标签域与 cache 标签的值也相同,则 cache 命中(hit);否则,称作 cache 失效(miss)。

在 cache 失效的情况下,控制器从主存中拷贝整个 cache 行到 cache 存储器中,为处理器核提供相应的代码或数据。这种拷贝整个 cache 行的操作被称作cache 行填充(cache line fill)。

在 cache 命中的情况下,控制器直接从 cache 存储器中为处理器核提供数据和代码。控制器使用数据索引域,在 cache 行中选择实际的代码或数据,并将其提供给处理器内核。

12.2.3 cache 与主存的关系

对 cache 存储器的基本结构和 cache 控制器的工作原理有了一定的了解之后,就可以来讨论 cache 与主存的关系了。

图 12.5 显示了主存中的部分内容是如何被临时存放在 cache 存储器中的。此图所示为最简单的 cache 形式——直接映射 cache。在一个直接映射 cache 中,主存中的每个地址都对应 cache 存储器中惟一的一行。由于主存的容量要远远大于 cache 存储器,所以在主存中有很多地址被映射到同一个 cache 行。由图 12.5 可以看出这种关系,所有以 0x824 结尾的内存地址都映射在同一 cache 行。

图 12.4 中介绍的地址信息的 3 个域:标签域、组索引域和数据索引域,在图 12.5 中仍然可以体现出来。组索引域(set index)可以确切地指出所有以 0x824 结尾的内存地址在 cache 中所惟一对应的存储地址;数据索引域可以确定字、半字或者字节在该 cache 行中的位置,在本例中是 cache 行中的第二个字;标签域则用来与 cache 行中的 cache-tag 相比较。

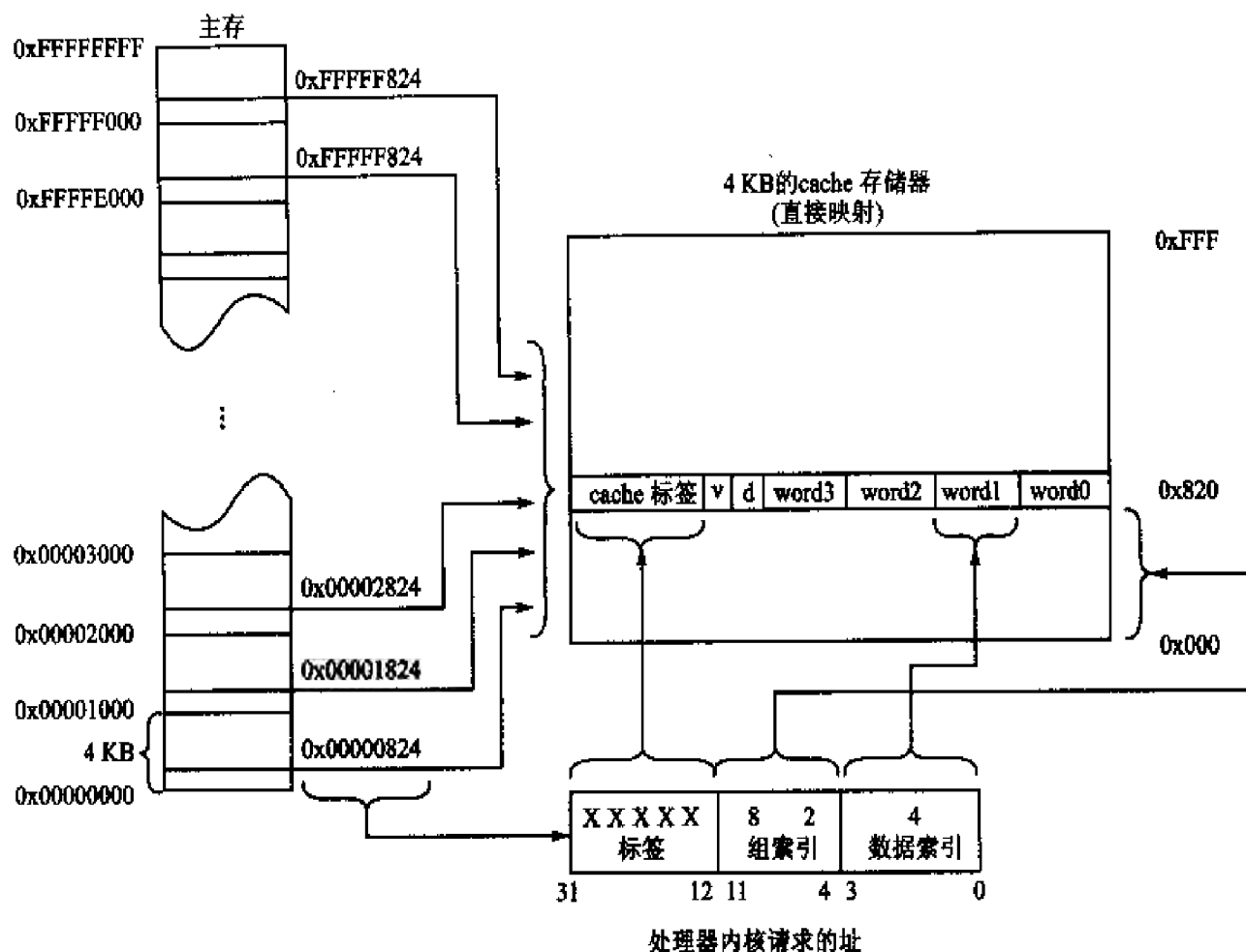


图 12.5 主存与直接映射 cache 的映射

在本例中,cache 中的每一行都对应着 100 万个内存地址。在任一给定时刻,这些地址中只有一个地址的内容可以出现在 cache 存储器中。将主存地址的标签域与 cache 行中的 cache 标签相比较,可以确定该 cache 行中是存储了处理器所要访问的主存单元数据,还是存储了另外一个地址以 0x824 结尾的主存单元数据。

在 cache 行被填充时,cache 控制器可以在向 cache 中搬运数据的同时,将数据传送到处理器内核中,这被称作数据流注(data streaming)。数据流注允许处理器一边执行程序,cache 控制器一边向相应 cache 行中搬运剩余的数据和代码。

如果在某一 cache 行中的数据虽然是有效的,但是与之对应的是主存中其它的地址块,而非处理器所要求的地址,那么整个 cache 行中内容将被删除,并被替换为与处理器内核所要求的地址相对应的 cache 行。这种移动一个有效 cache 行的过程,是 cache 失效处理的一部分,被称作“替换”,即将 cache 行中的内容返回到所对应的主存单元中,留出空间给需要加载到 cache 的新的数据。

ARM 嵌入式系统开发

直接映射 cache 是一种简单的解决方法,但这种设计使每个主存块在 cache 中只有一个特定的行可以存放。如果程序同时用到对应于 cache 同一行的 2 个主存块,那么就会发生冲突,冲突的结果就是导致 cache 行的频繁置换。这就是关于直接映射 cache 的颠簸 (thrashing) 问题——cache 存储器中同一位置的软件冲突。

图 12.6 所示为一个简单的软件循环在 cache 中频繁置换的过程。该程序在一个 do-while 循环中反复调用 2 个子程序,每个子程序都拥有相同的组索引域地址,所以这 2 个子过程(程序)在主存中的物理地址会映射到 cache 中相同的行。当第一次执行该循环并执行到子程序 A 时,A 被调入到 cache 行中。接下来当执行到子程序 B 时,A 所在的 cache 行被替换,同时 B 被放入该行,并开始执行。当第二次循环执行到 A 时,A 又把 B 从 cache 中替换出来,接着 B 再把 A 置换出去。重复的 cache 失效导致 cache 控制器连续不断地将当前不用的过程置换出 cache,这就是 cache 颠簸。

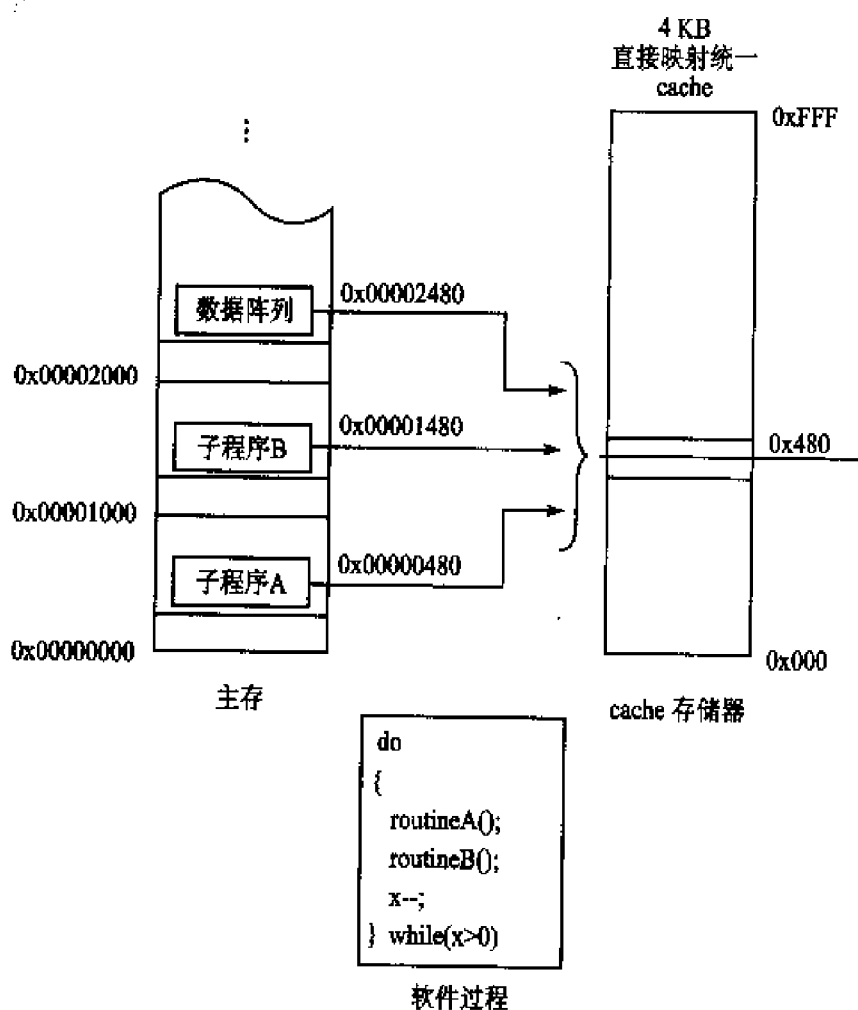


图 12.6 cache 颠簸 (thrashing): 在直接映射 cache 中 2 个函数相互替换

12.2.4 组相联

某些 cache 使用其它的设计方式,可以减少 cache 的颠簸频率(见图 12.7)。这种设计改变了 cache 的构造:将 cache 存储器分成了一些相同容量的小单元,称作路(*way*)。图 12.7 所示仍旧是一个 4 KB 的 cache,但是与前面所讲的 cache 不同的是,一个组索引域对应于多个 cache 行,即在每一路里都有一个 cache 行与之对应。前面所讲的 4 KB 容量的 cache 分为 256 行,现在把 cache 分成 4 路,每路有 64 个 cache 行。组索引域相同的 4 个 cache 行被称作处于同一个组(*set*)里,这也是组索引的命名的由来。

拥有相同组索引的 cache 行称为组相联的(*set associative*)。主存中的数据或者代码块可以在不影响程序执行的情况下被分配到组相联的任意一路中。换句话说,将数据或者代码存入 cache 行中的操作不会影响程序的执行。当主存中 2 个顺序的块被置换到 cache 中时,可以被放在同一路的连续 cache 行中,也可以被放在不同路中。需要注意的是,主存中特定位置的代码或者数据被读入到 cache 时,可以被存放在同一个组的任意 cache 行中。在 cache 的同一个组当中,数据放置的位置具有排他性,可以防止同样的数据被重复放在一个组的不同的 cache 行。

在 4 个 way 的组相联 cache 中,主存到 cache 的映射与以前有所不同(见图 12.8)。主存中的一个地址现在可以映射到 cache 中的 4 个不同地址。虽然图 12.5 与图 12.8 所示都是 4 KB 的 cache,但它们当中还是有很多值得注意的差异。

在图 12.8 中,tag 域比以前多了 2 位,而同时组索引域比以前少了 2 位。这意味着主存中的 400 万地址映射到 cache 一个组中的 4 个 cache 行中,而不是 100 万主存地址映射到一个 cache 位置。

现在,主存映射到 cache 中的大小是 1 KB 而不是 4 KB。这意味着将数据块映射到同一组的 cache 行的可能性比以前增加了 4 倍,同时一个 cache 行被替换的概率也减小为原来的 1/4。

如果图 12.6 中的示例程序代码在图 12.8 中的 4 个 way 的组相联 cache 中运行,那么 cache 失效的可能性会大大降低,子程序 A、子程序 B 和数据组会分别被存储在一个组的 4 个可能位置中的一个。当然,这是假设每个子程序和数据组的大小要小于从主存中映射过来的 1 KB 空间。

提高相联度

随着 cache 控制器的相联度提高,冲突的可能性减小了。理想的目标是,尽量提高组相联程度,使主存地址能够映射到任意 cache 行。这样的 cache 被称为全相联 cache。然而,随着相联度的提高,与之相匹配的硬件的复杂程度也在提高。硬件设计者提高 cache 相联

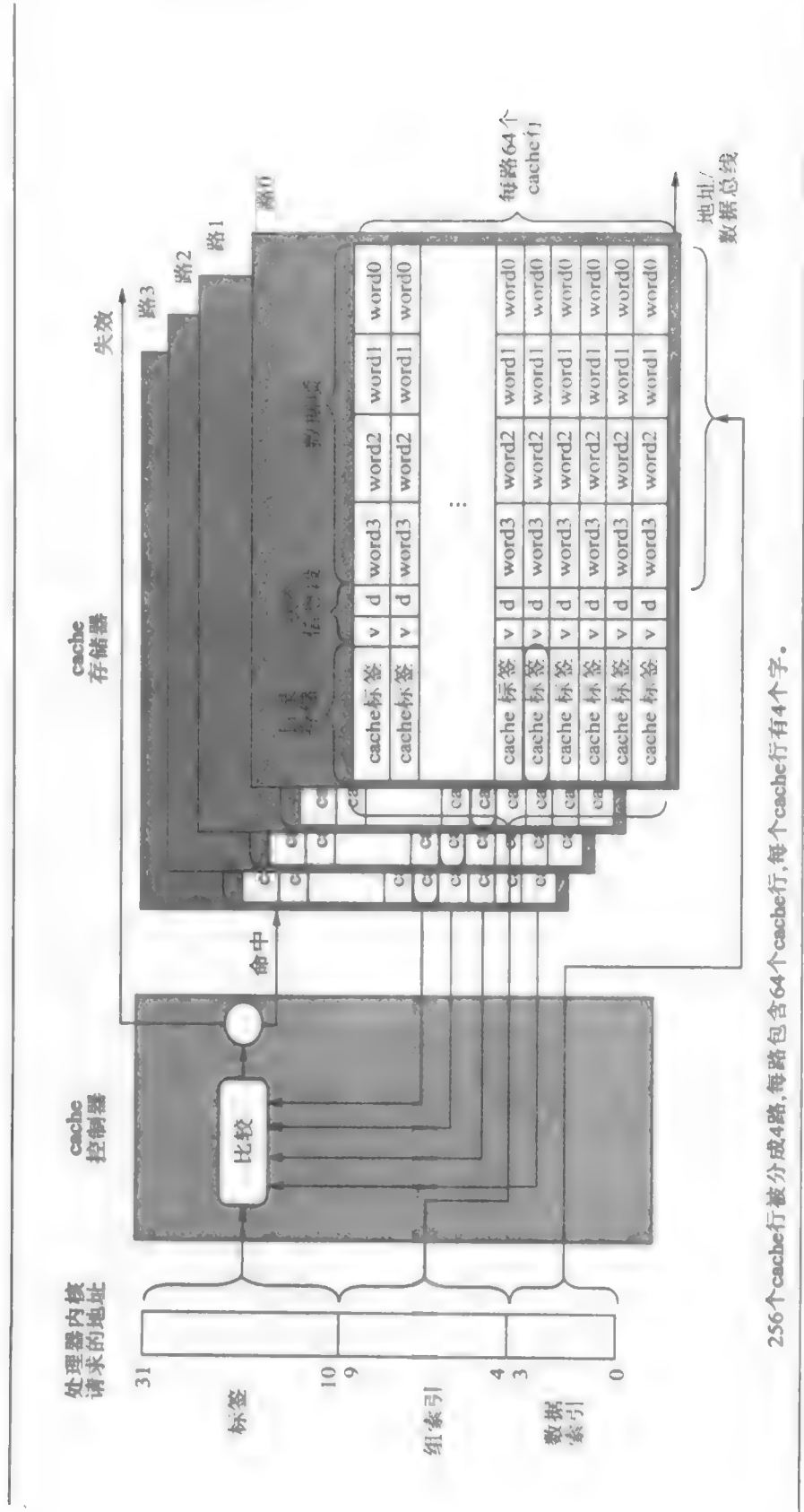


图 12.7 一个 4 KB 四路组相联 cache

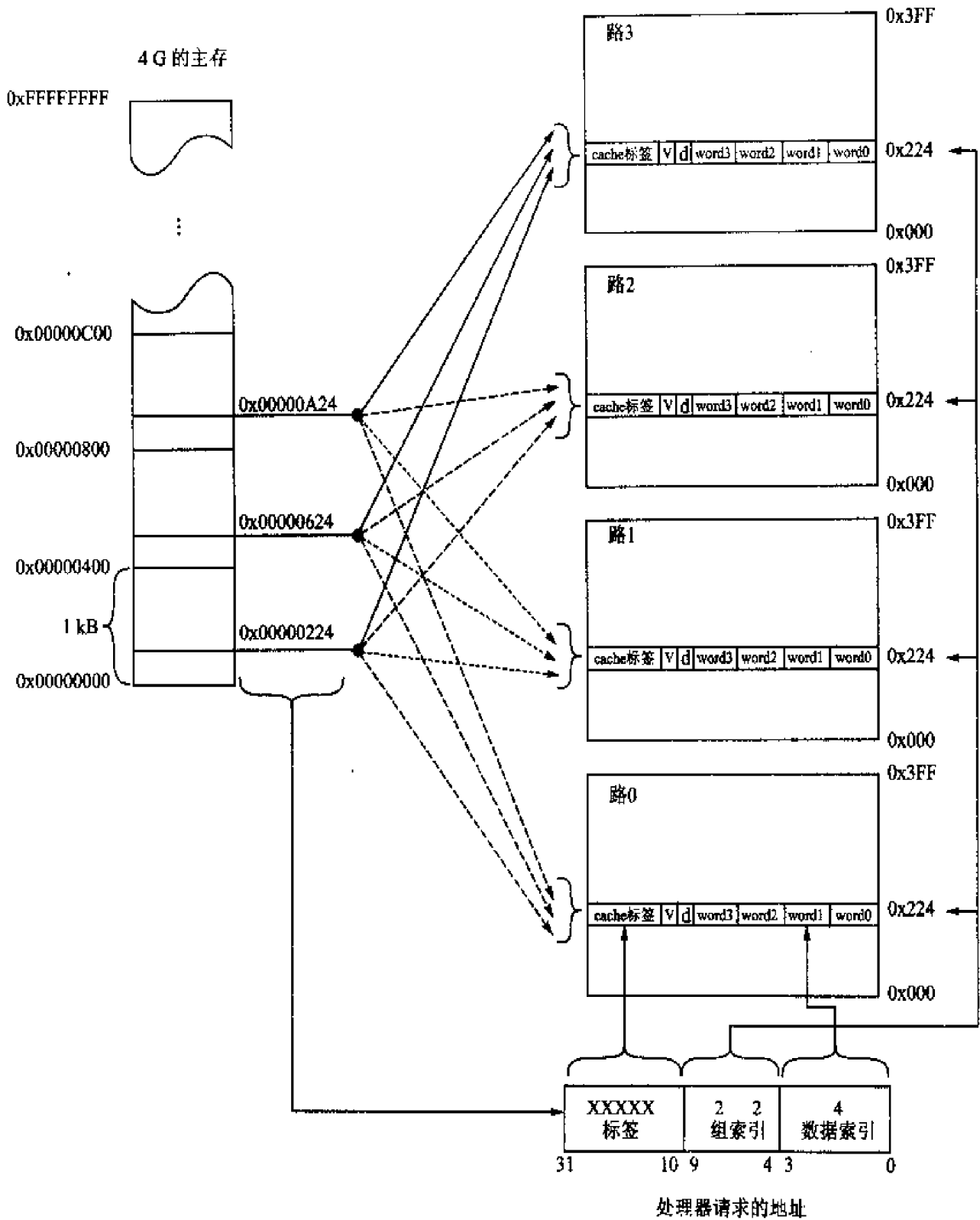


图 12.8 主存与 4 路组相联 cache 的映射

度的一种方法就是使用内容寻址存储器 CAM(Content Addressable Memory)。

CAM 使用一组比较器,以比较输入的标签地址和存储在每一个有效 cache 行中的 cache tag。CAM 采用了与 RAM 相反的工作方式:RAM 是得到一个地址后再给出数据;而 CAM 则是在检测到给定的数据值在存储器中后,再给出该数据的地址。使用 CAM 允许同时比较更多的 cache tag,从而增加了可以包含在一个组中的 cache 行数。

在 ARM920T 和 ARM940T 处理器核中,ARM 使用了 CAM 来定位 cache-tag。ARM920T 和 ARM940T 中的 cache 是 64 路组相联的。图 12.9 所示为 ARM940T 的 cache 结构图。cache 控制器把地址标签(address tag)作为 CAM 的输入,它的输出选择了包含有效 cache 行的路(way)。

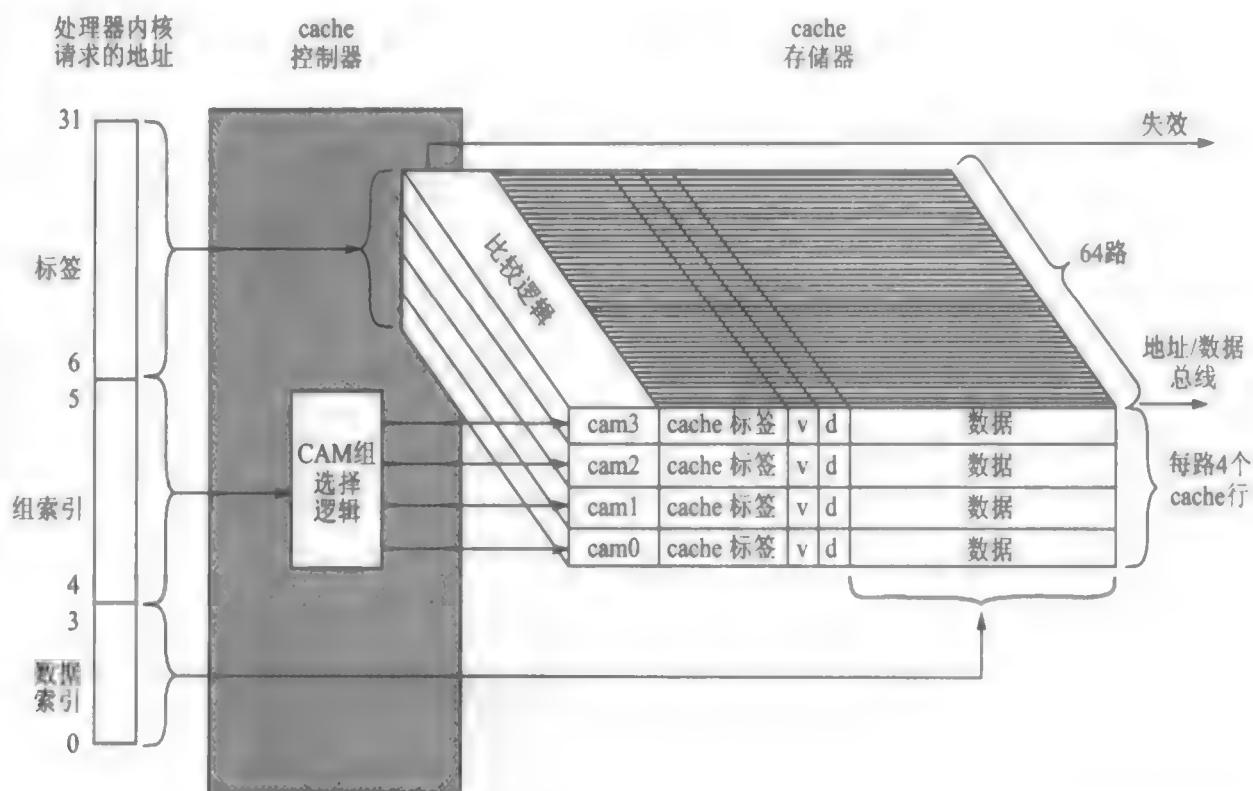


图 12.9 ARM940T——使用 CAM 的 4 KB 64 路组相联 D-cache

访问地址的 tag 部分被作为 4 个 CAM 的输入,输入标签同时与存储在 64 路中的所有 cache 标签相比较。如果有一个匹配,那么数据就由 cache 存储器提供;如果没有匹配,存储器控制器就会产生一个失效(miss)信号。

控制器使用组索引位(set index)来选择 4 个 CAM 中的一个。被选中的 CAM 会在 cache 存储器中选择一个 cache 行,该地址的数据索引部分(data index)在该 cache 行中选择

出所需要的字、半字或者字节。

12.2.5 写缓冲器

写缓冲器是一个容量非常小的高速 FIFO 存储缓冲器,用来临时存放处理器将要写入到主存中的数据。在没有写缓冲器的系统中,处理器直接写数据到主存中。在带有写缓冲器的系统中,数据先高速写入 FIFO,然后再写入低速的主存中。写缓冲器缩短了写小块序列数据到主存时的处理器时间。写缓冲器中的 FIFO 存储器在存储层次中,与 L1 cache 处于相同的层次(见图 12.1)。

写缓冲器的效率依赖于主存写的次数与执行指令数的比例。在给定的时间间隔中,若主存写的次数比较少,或者写操作与其它操作指令有足够的间隔,那么写缓冲器一般就不会满。在写缓冲器不满的情况下,运行程序可以使用寄存器操作来连续执行超出 cache 的访问。此时,使用 cache 进行读/写,使用写缓冲器来临时存放被替换出的 cache 行内容,并马上被写入主存。

写缓冲器同时还改善了 cache 的性能,这体现在 cache 行被替换时。当 cache 控制器要替换出一个脏的 cache 行时,它只将该 cache 行放入写缓冲器中,而不写入主存。这样,可以更快地填充新的 cache 行数据,处理器就可以继续从 cache 存储器中读/写数据。

写缓冲器中的数据在没有被写入主存之前,是不能被读取的。同样,被替换的 cache 行在写缓冲器中时也不能进行读操作。这也是为什么写缓冲器的 FIFO 深度通常比较小的原因之一,一般只有几个 cache 行的深度。

有些写缓冲器并不是严格的 FIFO 缓冲器。例如,ARM10 系列支持接合(*coalescing*)——把写操作合并到一个单一的 cache 行。也就是说,写缓冲器会把新的数值(如果它们表示的是主存中同一个数据块)合并到一个在写缓冲器中已存在的 cache 行。接合又被称作写合并、写联合或写结合等。

12.2.6 cache 效率的衡量

有 2 个性能指标可以衡量一个程序的 cache 效率:cache 命中率(hit rate)和 cache 失效率(miss rate)。在给定的时间间隔内,cache 命中的次数与总的存储器请求次数的比值被称作命中率。命中率可以用下面的百分数来表示:

$$\text{命中率} = \left(\frac{\text{cache 命中次数}}{\text{存储器请求次数}} \right) \times 100\%$$

失效率与命中率形式相似:在给定的时间间隔内,cache 失效的总次数除以总的存储器

ARM 嵌入式系统开发

请求次数所得的百分数。失效率与命中率之和等于 100。

命中率和失效率可以衡量数据的读、写,或者同时衡量读、写两者的效率。也就是说,这 2 个性能指标可以从几个方面来描述系统的性能情况。例如:可以计算读数据的命中率、写数据的命中率,或者其它方面操作的命中率和失效率等等。

另外两个衡量 cache 性能的指标是命中时间(hit time)和失效开销(miss penalty)。命中时间是指处理器访问 cache 中数据时所需要的时间。失效开销是指处理器从主存中装载一个 cache 行数据到 cache 所需要的时间。

12.3 cache 策略

有 3 种可以决定 cache 操作的策略:写策略、替换策略及分配策略。cache 的写策略决定了处理器执行写操作时数据存放的位置。替换策略在 cache 失效的情况下,决定选择被替换出主存的 cache 行。分配策略决定 cache 控制器在何时将要分配 cache 行。

12.3.1 写策略——直写法或回写法

处理器核向存储器写数据时,cache 控制器可以有 2 种可选择的写策略。它可以同时向 cache 行和相应的主存位置中写入数据,将存储在 2 个位置上的数据一起更新,这种做法被称为直写法(writethrough)。另外,它也可以只把数据写入相应的 cache 行,而不写入主存,只有当相应 cache 行被替换或清理 cache 行时,才被写入主存,这种做法被称为回写法(writeback)。

12.3.1.1 直写法

如果 cache 控制器使用直写策略,那么处理器核写 cache 命中时,将同时修改 cache 和主存中的内容,以确保 cache 和主存数据的一致性。在这种策略下,处理器核在每次写 cache 时也要写相应的主存单元。由于要访问主存,直写法的速度比回写法要慢一些。

12.3.1.2 回写法

如果 cache 控制器使用回写策略,那么处理器核写 cache 命中时,只向 cache 存储器写数据,而不立即写入主存。这样,主存块与相应的 cache 行数据有可能不一致。cache 中的数据是最新的,而主存中的数据可能是较早的、没有被更新过的。

配置成回写法的 cache 要使用到 cache 行的状态信息块中的一个或多个脏位(dirty bit)。当回写 cache 控制器向 cache 存储器中某一行写入数据时,它会将脏位设置为 1。如果控制器内核此后访问该 cache 行,那么通过脏位的状态就可以知道该 cache 行中含有主存

中没有的数据。如果 cache 控制器要将一个脏位被置位的 cache 行替换出 cache 存储器,那么该 cache 行数据会自动被写到主存单元中去。控制器通过这种方式来防止只存在于 cache 中而主存中没有的重要信息的丢失。

当一个程序频繁使用某些临时的局部变量时,由于这些变量是临时的,所以根本用不着被写到主存中去。此时回写法 cache 优于直写法 cache。例如,当寄存器文件没有足够的寄存器来存放临时局部变量时,就会导致部分变量溢出到一个 cache 堆栈中,这些临时变量就不需要写入主存。

12.3.2 cache 行替换策略

当一个 cache 访问失效时,cache 控制器必须从当前有效的组中选择一个 cache 行来存储从主存中取得的新信息。被选中替换的 cache 行被称为丢弃者(*victim*)。如果丢弃者中包含有效的脏数据,那么在该 cache 行被写入新数据之前,控制器必须把该行中的数据写到主存。选择和替换丢弃 cache 行的过程被称作淘汰(*eviction*)。

cache 控制器选择下一个丢弃 cache 行的策略被称为替换策略。cache 替换策略从当前有效的相联组中选择一个 cache 行,即它选择一路(*way*)用于下一次 cache 行替换。总的来说,组索引域在各个 way 中选择可用的一组 cache 行;而替换策略决定在该组中的哪一个 cache 行被新的数据所替换。

带 cache 的 ARM 核支持两种替换策略:伪随机替换法和轮转法。

- 轮转法又叫循环替换,这种方法只是简单地将当前分配 cache 行的下一行作为被替换的行。它所采用的选择算法使用了连续加 1 的丢弃计数器,该计数器在每一次 cache 控制器分配新的 cache 行时都会自动加 1。当丢弃计数器计数达到最大值时,就被复位成预先定义好的一个基值。
- 伪随机替换法从特定的位置上随机地选出一行替换出去。该算法使用了非连续增加的丢弃计数器,控制器随机产生一个增加值,并将该增加值加到丢弃计数器上。同样,当丢弃计数器计数达到最大值时,会被复位成预先定义好的一个基值。

大多数 ARM 核都支持这两种替换策略(表 12.1 详尽列出了各种 ARM 核及其所支持的策略)。相比之下,轮转法替换策略有更好的可预测性,容易预测最坏情况下 cache 的性能,这在嵌入式系统中是很必要的;然而,轮转法替换策略在存储器访问发生很小的变化时,有可能造成 cache 性能有较大的变化。可以从例 12.1 看出这种性能上的改变。

表 12.1 带 cache 的 ARM 核所使用的策略

内 核	写策略	替换策略	分配策略
ARM720T	直写法	随 机	读分配
ARM740T	直写法	随 机	读分配
ARM920T	直写法,回写法	随机,轮转法	读分配
ARM940T	直写法,回写法	随 机	读分配
ARM926EJS	直写法,回写法	随机,轮转法	读分配
ARM946E	直写法,回写法	随机,轮转法	读分配
ARM10202E	直写法,回写法	随机,轮转法	读分配
ARM1026EJS	直写法,回写法	随机,轮转法	读分配
Intel StrongARM	回写法	轮转法	读分配
Intel XScale	直写法,回写法	轮转法	读、写分配

【例 12.1】 说明分别使用轮转和伪随机替换策略运行一个程序所耗费的时间。

测试程序 cache_RRtest 使用 C 库文件 header.h 中的时钟函数来计算时间。程序先使用轮转策略进行时间测试,然后使用随机策略进行了相同的测试。

测试程序 readSet 是基于 ARM940T 的。该程序使用轮转替换策略,并故意显示了一种 cache 性能的最坏的突变。

```
#include <stdio.h>
#include <time.h>

void cache_RRtest(int times,int numset)
{
    clock_t count;

    printf("Round Robin test size = %d\r\n", numset);
    enableRoundRobin();
    cleanFlushCache();
    count = clock();
    readSet(times,numset);
    count = clock() - count;
    printf("Round Robin enabled = %.2f seconds\r\n",
           (float)count/CLOCKS_PER_SEC);
    enableRandom();
}
```



```

cleanFlushCache();
count = clock();
readSet(times, numset);
count = clock() - count;
printf("Random enabled = %.2f seconds\r\n\r\n",
       (float)count/CLOCKS_PER_SEC);
}

int readSet( int times, int numset)
{
    int setcount, value;
    volatile int * newstart;
    volatile int * start = (int *)0x20000;

    __asm
    {
        timesloop:
            MOV     newstart, start
            MOV     setcount, numset
        setloop:
            LDR     value, [newstart, #0];
            ADD     newstart, newstart, #0x40;
            SUBS     setcount, setcount, #1;
            BNE     setloop;
            SUBS     times, times, #1;
            BNE     timesloop;
    }
    return value;
}

```

readSet 例程用来填充 cache 的一个组(set)。这个函数有 2 个参数:第 1 个 times,运行测试循环的次数,这个值增加了运行测试所花费的时间;第 2 个 numset,即在一个组(set)中需要读取的数据的个数,它决定程序要读取多少 cache 行到同一个组中。通过一个循环向 cache 的组中填写数据,该循环使用 LDR 指令在主存的某个位置上读取一个数值,并且每循环一次将地址指针加上 16 个字(即 64 字节)。在 ARM940T 中,设置 numset 的值为 64,可以填满一个组的所有可用的 cache 行。ARM940T 的一路(way)有 16 个字,每组有 64 个 cache 行。

以下代码使用 2 个不同的组大小进行了 2 次调用,以测试轮转替换策略。第 1 次用 64

个条目(entries)充填一个 cache 组;第 2 次用 65 个条目(entries)充填一个 cache 组。

```
unsigned int times = 0x10000;
unsigned int numset = 64;

cache_RRtest (times, numset);
numset = 65;
cache_RRtest ( times,numset);
```

下面是 2 个测试运行的输出结果。测试程序在 ARM940T 处理器核、ARM ADS 1.2 ARMulator 环境下运行。处理器核的时钟为 50 MHz,非顺序访问的存储器读时间为 100 ns,顺序访问为 50 ns。需要注意的是使用轮转策略读取 65 个组数值时的时间变化。

```
Round Robin test size = 64
Round Robin enabled = 0.51 seconds
Radom enabled = 0.51 seconds
Round Robin test size = 65
Round Robin enabled = 2.56 seconds
Radom enabled    = 0.58 seconds
```

这是个比较极端的例子,但是它反映了使用伪随机替换策略和轮转法替换策略的不同之处。

还有一种替换策略是最近最少使用策略 LRU(Least Recently Used)。该策略记录 cache 行的使用情况,将近期内最长时间未被访问过的 cache 行替换出 cache。

带 cache 的 ARM 核不支持最近最少使用策略,但是有些 ARM 的半导体合作伙伴在其制造的芯片中将自己的 cache 加到不带 cache 的 ARM 核中。所以有些基于 ARM 的产品支持最近最少使用策略。

12.3.3 cache 失效时的分配策略

在 cache 失效发生时,ARM 的 cache 可以采取两种策略来分配 cache 行;第一种叫做读操作分配(read-allocate)策略;第二种叫做读/写操作分配(read-write-allocate)策略。

如果 cache 未命中,那么对于读操作分配策略,只有进行存储器读操作时,才分配 cache 行。如果被替换的 cache 行包含有效数据,那么在该行被新的数据填充之前,要先把其原来的内容写入主存中去。

采用读操作分配策略时,存储器写操作并不会更新 cache 存储器中的内容,除非相关的 cache 行恰好是前一个主存读操作刚刚分配的。如果这个 cache 行中包含有效数据,那么在采用直写策略时,写操作更新 cache 的同时,还会更新主存中的相应内容。如果写操作的对

象不在 cache 中,那么写操作只更新主存中的相应内容。

采用读/写操作分配策略时,不管是存储器读操作,还是存储器写操作,在 cache 未命中时都将分配 cache 行。对于主存的任何写操作,如果操作对象不在 cache 中,那么 cache 控制器也会分配一个新的 cache 行,并把主存中的相应内容填充到该 cache 行。对于存储器读操作,cache 控制器运用读操作分配策略。

当内核进行数据写操作时,如果 cache 未命中,那么 cache 控制器将会分配一个 cache 行。如果被替换出的 cache 行中包含有效数据(valid data),那么在主存将新的内容放入该 cache 行之前,控制器会将该行的内容先写入主存。如果该行的数据无效,那么它将直接被主存中的新数据覆盖。分配的 cache 行被填充后,控制器才将内核数据写到该 cache 行的相关位置。对于直写 cache,数据将会同时被写入到主存中。^{*}

ARM7,ARM9 和 ARM10 的内核在 cache 失效时使用读操作分配策略,Intel XScale 在 cache 失效时可以同时支持读操作分配和写操作分配策略。表 12.1 列出了各种 cache 核所支持的各种策略。

12.4 协处理器 15 与 cache

协处理器 15(CP15)的一些寄存器是专门用来配置和控制带 cache 的 ARM 内核的。表 12.2 列出了控制 cache 配置的协处理器 15 寄存器。CP15 的主寄存器 c7 和 c9 控制着 cache 的设置和操作。辅寄存器 CP15;c7 是只写的,控制清除或清理 cache;CP15;c9 定义将被替换的丢弃者(victim)指针的基地址,该基地址决定了锁定在 cache 中的代码和数据的行数。后续章节中,将会详细讨论有关命令。有关协处理器 15 的命令和句法,可参见 3.5.2 小节。

还有其它一些影响 cache 操作的 CP15 寄存器,这些寄存器的定义依赖于内核。有关这些寄存器的内容将在第 13 章的 13.2.3 小节和 13.2.4 小节的 MPU 初始化,以及第 14 章的 14.3.6 小节的 MMU 初始化中介绍。

在接下来的几节中,将使用表 12.2 中列出的 CP15 寄存器编写清理和清除 cache 的示例程序,并在 cache 中锁定代码和数据。控制系统通常调用这些子程序作为存储管理的一部分。

^{*} 只要分配了 cache 行,即使是内核执行写操作,cache 控制器也先要把主存的相关内容复制到所分配的 cache 行,内核数据才能写入该 cache 行中。——译者注

表 12.2 配置和控制 cache 操作的协处理器 15 寄存器

功 能	主寄存器	辅寄存器	操作码(Opcod)2
清理和清除 cache	c7	c5, c6, c7, c10, c13, c14	0, 1, 2
排空写缓冲器	c7	c10	4
cache 锁定	c9	c0	0, 1
轮转法替换	c15	c0	0

12.5 清除和清理 cache

ARM 使用术语**清除**(flush)和**清理**(clean)表示对 cache 的两种基本操作。

清除 cache 的意思是清除 cache 中存储的全部数据。对处理器而言,清除操作只要清零相应 cache 行的有效位即可。当存储器配置上有变化时,整体或部分 cache 可能需要进行清除操作。有时也用术语**作废**(invalidate)来替代术语“清除”。然而,对于采用回写策略的 D-cache,就需要使用**清理**(clean)操作。*

清理 cache 的意思是把脏的(即被改写过的)cache 行强制写到主存,并把 cache 行中的脏位(污染位)清零。**清理 cache** 可以重建 cache 与主存之间的一致性,它只在使用回写策略的 D-cache 上。

改变系统的存储器配置可能需要执行清除和清理 cache 的操作。访问权限、cache 和缓冲策略的变化或者重新映射虚拟地址等操作都需要清理或清除 cache。

在分离 cache 中执行自修改代码之前,cache 也需要执行清理和清除操作。自修改代码包括将代码简单地从一个地方拷贝到另一个地方。清理和清除操作是由两种可能的情况引起的:第一,自修改代码可能被承载在 D-cache 中,因此,不可能作为一条指令从主存中进行加载;第二, I-cache 中现存的指令可能会屏蔽写到主存中的新指令。

如果 cache 使用回写策略并且自修改代码被写入主存中,那么第一步就是将指令以数据块的形式写到主存中的某处;稍后,程序跳转到主存中,以指令流的形式从主存中的该处开始执行。其中,当代码作为数据写入到主存中时,如果 cache 存储器中代表自修改代码被写入的主存位置的 cache 行有效,那么代码有可能会被写入到 cache 中(没有写入主存)。这些 cache 行会被拷贝到 D-cache,而不是被拷贝到主存。如果发生了这种情况,那么当程序跳转到自修改代码所在地方(主存某处)时,就会执行原来数据表示的代码,因为自修改代码

* 这里没有把“flush”译成“刷新”,因为“刷新”一词有新内容重新填入的意思。——译者注

此时实际上还在 D-cache 中。为了防止这种情况发生,可以进行 D-cache 的清理操作,把指令代码强制作为数据存到主存中,从而这些数据就可以作为指令流从主存中被读取出来。

D-cache 被清理后,新的指令就被写入到主存中。但是,I-cache 中可能会有有效 cache 行存储新数据(代码)地址对应的指令。接下来,在新代码所在的地址读取指令时,仍然会得到 I-cache 中的老代码,而不是主存中的新代码。清除 I-cache 可以防止这种情况的发生。

12.5.1 清除 cache

清除 cache,即使 cache 中的内容无效。如果 cache 使用回写策略,那么在清除之前应该清理 cache,以防止由于清除操作使数据丢失。

有 3 个 CP15:c7 命令可以在 cache 中执行清除操作:第 1 个清除整个 cache;第 2 个只清除 I-cache;第 3 个只清除 D-cache。这些命令以及支持它们的内核见表 12.3。对于这 3 个 MCR 指令,处理器内核寄存器 Rd 的值应该为零。

表 12.3 清除 cache 的 CP15:c7:Cm 命令

命 令	MCR 指令	支持的内核
清除 cache	MCR p15,0,Rd,c7,c7,0	ARM720T,ARM920T,ARM922T,ARM926EJ-S ARM1022E,ARM1026EJ-S,StrongARM,XScale
清除数据 cache	MCR p15,0,Rd,c7,c6,0	ARM920T,ARM922T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S StrongARM,XScale
清除指令 cache	MCR p15,0,Rd,c7,c5,0	ARM920T,ARM922T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S StrongARM,XScale

例 12.2 显示了如何使用这 3 条指令清除 cache。读者可以直接使用本例,也可以根据系统需要稍做修改。示例中包含一个产生 3 个例程的宏(使用宏的详细信息见附录 A):

- *flushICache* 清除 I-cache;
- *flushDCache* 清除 D-cache;
- *flushCache* 清除 I-cache 和 D-cache。

这些例程没有输入参数,使用以下的原型从 C 程序中调用:

```
void flushCache (void);      /* 清除全部 cache */
void flushDCache (void);    /* 清除 D-cache */
void flushICache (void);    /* 清除 I-cache */
```

ARM 嵌入式系统开发

【例 12.2】 本例首先根据所支持的不同命令将内核分组。

这里使用一个名为 CACHEFLUSH 的宏来创建例程。这个宏首先设置内核寄存器，将 0 写入 CP15:c7:Cm。接下来根据需要的 cache 操作的类型和不同的内核，插入特定的 MCR 指令。

```

IF {CPU} = "ARM720T"      ;LOR; \
{CPU} = "ARM920T"        ;LOR; \
{CPU} = "ARM922T"        ;LOR; \
{CPU} = "ARM926EJ - S"   ;LOR; \
{CPU} = "ARM940T"        ;LOR; \
{CPU} = "ARM946E - S"    ;LOR; \
{CPU} = "ARM1022E"       ;LOR; \
{CPU} = "ARM1026EJ - S"  ;LOR; \
{CPU} = "SA - 110"       ;LOR; \
{CPU} = "XSCALE"

c7f      RN 0 ;CP15:c7 格式的寄存器
        MACRO
        CACHEFLUSH $ op

        MOV      c7f, #0
        IF      "$ op" = "Icache"
            MCR    p15,0,c7f,c7,c5,0      ;清除 I-cache
        ENDIF
        IF      "$ op" = "Dcache"
            MCR    p15,0,c7f,c7,c6,0      ;清除 D-cache
        ENDIF
        IF      "$ op" = "IDcache"
            IF {CPU} = "ARM940T" ;LOR; \
               {CPU} = "ARM946E - S"
                MCR    p15,0,c7f,c7,c5,0      ;清除 I-cache
                MCR    p15,0,c7f,c7,c6,0      ;清除 D-cache
            ELSE
                MCR    p15,0,c7f,c7,c7,0      ;清除 I-cache 和 D-cache
            ENDIF
        ENDIF
        MOV      pc, lr
        MEND

```

```

        IF {CPU} = "ARM720T"
            EXPORT flushCache
flushCache
            CACHEFLUSH IDcache
        ELSE
            EXPORT flushCache
            EXPORT flushICache
            EXPORT flushDCache
flushCache
            CACHEFLUSH IDcache
flushICache
            CACHEFLUSH Icache
flushDCache
            CACHEFLUSH Dcache
        ENDIF
    ENDIF

```

最后,多次使用宏创建例程。ARM720T 具有指令-数据统一 cache,所以只使用 flush-Cache 例程即可。对于其它内核,3 次使用宏来创建 3 个例程。

这个例子比大多数实现需求包含更多的代码。但是它给出了一个支持当前所有 ARM 内核的详尽程序。读者可以根据特定的 ARM 内核在例 12.2 的基础上编写较简单的程序。这里以 ARM926EJ-S 为例,示范如何从例 12.2 中提取 3 个例程。重新编写的代码如下:

```

EXPORT flushCache926
EXPORT flushICache926
EXPORT flushDCache926
C7f RN 0 ; CP15,c7 格式的寄存器
flushCache926
    MCR p15,0,c7f,c7,c7,0 ;清除 I-cach 和 D-cache
    MOV pc, lr
flushICache926
    MCR p15,0,c7f,c7,c5,0 ;清除 I-cach
    MOV pc, lr
flushDCache926
    MCR p15,0,c7f,c7,c6,0 ;清除 D-cach
    MOV pc, lr

```

如果用 C 语言编写,则可使代码更加简单,并写成能被放在 include 文件中的内嵌函数

的形式。内嵌函数如下：

```
__inline void flushCache926 (void)
{
    unsigned int c7format = 0;
    __asm{MCR p15,0,c7format,c7,c7,0};    /* 清除 I-cache 和 D-cache */
}

__inline void flushDCache926 (void)
{
    unsigned int c7format = 0;
    __asm{MCR p15,0,c7format,c7,c6,0};    /* 清除 D-cache */
}

__inline void flushICache926 (void)
{
    unsigned int c7format = 0;
    __asm{MCR p15,0,c7format,c7,c5,0};    /* 清除 I-cache */
}
```

本章中的其余例子是使用 ARM 汇编语言编写的,并适用于当前所有内核。相同的提取过程也适用于其它例子。

12.5.2 清理 cache

清理 cache 即执行指令,命令 cache 控制器将所有带有脏位的 D-cache 行写入到主存中去。在这个过程中,cache 行中的脏位将被清除。清理 cache 操作可以重建 cache 和主存之间的数据的一致性。此操作仅适用于使用回写策略的 D-cache。

术语回写(*writeback*)和回拷(*copyback*)在一些地方也通常表示清理(*clean*)的意思。这些术语与描述 cache 写策略的用词相近,但在这种情况下,它们描述的是对 cache 存储器所作的操作。在非 ARM 系统中,术语刷新(*flush*)往往与 ARM 系统中清理(*clean*)意思相同。

12.5.3 清理 D-cache

到本书截稿为止,有 3 种清理 D-cache 的方法(见表 12.4)。具体的处理方法与处理器有很大关系,因为不同的处理器内核有各自不同的指令来清理 D-cache。

表 12.4 清理 D-cache 的方法

方 法	示 例	处 理 器
路和组索引寻址	例 12.3	ARM920T, ARM922T, ARM926EJ-S, ARM940T, ARM946E-S, ARM1022E, ARM1026EJ-S
测试清理(test-clean)	例 12.4	ARM926EJ-S, ARM1026EJ-S
读某一特定存储器块的特殊分配策略	例 12.5	XScale, SA-110

虽然清理 cache 的方法有很多种,但在下面的例子中,会提供相同的过程调用,为所有的内核提供一致性的接口。这里给出了 3 个程序来清理整个 cache,cache 会在每次清理之前做一次写操作:

- *cleanDCache* 清理 D-cache;
- *cleanFlushDCache* 清理并清除 D-cache;
- *cleanFlushCache* 清理并清除 D-cache 和 I-cache。

CleanDCache, cleanFlushDCache 和 cleanFlushCache 三个过程不需要输入参数,并且可以在 C 程序中以下面的形式来调用。

```
Void cleanDCache(void);           /* 清理 D-cache */
Void cleanFlushDCache(void);      /* 清理并清除 D-cache */
Void cleanFlushCache(void);       /* 清理并清除 D-cache 和 I-cache */
```

在编写这些例子中的宏时,应使它们不需要做大量修改就可以支持尽可能多的 ARM 内核。为了达到这个目的,在本例中和本章的其它例子中,使用了一个通用的头文件,头文件名为 cache.h,见以下程序:

```
IF {CPU} = "ARM920T"
    CSIZE    EQU 14                ;cache size as 1 << CSIZE (16 K assumed)
    CLINE    EQU 5                ;cache line size in bytes as 1 << CLINE
    NWAY     EQU 6                ;set associativity = 1 << NWAY (64 way)
    I7SET    EQU 5                ;CP15 c7 st incrementer as 1 << ISET
    I7WAY    EQU 26               ;CP15 c7 way incrementer as 1 << SSET
    I9WAY    EQU 26               ;CP15 c9 way incrementer as 1 << SSET
ENDIF
IF {CPU} = "ARM922T"
    CSIZE    EQU 14                ;cache size as 1 << CSIZE (16 K assumed)
    CLINE    EQU 5                ;cache line size in bytes as 1 << CLINE
    NWAY     EQU 6                ;set associativity = 1 << NWAY (64 way)
```

```

I7SET      EQU 5
I7WAY      EQU 26
I9WAY      EQU 26
ENDIF

IF {CPU} = "ARM926EJ-S"

CSIZE      EQU 14                ;cache 大小为 1 << CSIZE(假定为 16K)
CLINE      EQU 5                ;cache 行的字节数为 1 << CLINE
NWAY       EQU 2                ;组相联度 = 1 << NWAY (4 路)
I7SET      EQU 4
I7WAY      EQU 30
ENDIF

IF {CPU} = "ARM940T"

CSIZE      EQU 12                ;cache 大小为 1 << CSIZE (4K)
CLINE      EQU 4                ;cache 行的字节数为 1 << CLINE
NWAY       EQU 6                ;组相联度 = 1 << NWAY (64 路)
I7SET      EQU 4
I7WAY      EQU 26
I9WAY      EQU 0
ENDIF

IF {CPU} = "ARM946E-S"

CSIZE      EQU 12                ;cache 大小为 1 << CSIZE (假定为 4 K)
CLINE      EQU 5                ;cache 行的字节数为 1 << CLINE
NWAY       EQU 2                ;组相联度 = 1 << NWAY (4 路)
I7SET      EQU 4
I7WAY      EQU 30
I9WAY      EQU 0
ENDIF

IF {CPU} = "ARM1022E"

CSIZE      EQU 14                ;cache 大小为 1 << CSIZE (16 K)
CLINE      EQU 5                ;cache 行的字节数为 1 << CLINE
NWAY       EQU 6                ;组相联度 = 1 << NWAY (64 路)
I7SET      EQU 5
I7WAY      EQU 26
I9WAY      EQU 26
ENDIF

IF {CPU} = "ARM1026EJ-S"

CSIZE      EQU 14                ;cache 大小为 1 << CSIZE (假定为 16 K)

```

```

CLINE    EQU 5                ;cache 行的字节数为 1 << CLINE
NWAY     EQU 2                ;组相联度 = 1 << NWAY (4 路)
I7SET    EQU 5
I7WAY    EQU 30
        ENDIF

        IF {CPU} = "SA-110"

CSIZE     EQU 14              ;cache 大小为 1 << CSIZE (16 K)
CLINE     EQU 5              ;cache 行的字节数为 1 << CLINE
NWAY      EQU 5              ;组相联度 = 1 << NWAY (4 路)
CleanAddressDcache EQU 0x8000
        ENDIF

        IF {CPU} = "XSCALE"

CSIZE     EQU 15              ;cache 大小为 1 << CSIZE (32 K)
CLINE     EQU 5              ;cache 行的字节数为 1 << CLINE
NWAY      EQU 5              ;组相联度 = 1 << NWAY (32 路)
MNWAY     EQU 1              ;迷你 D-cache 的组相联度 = 1 << MNWAY (2 路)
MCSIZE    EQU 11             ;迷你 cache 的大小为 1 << MCSIZE (2 K)
        ENDIF

, -----
SWAY      EQU (CSIZE - NWAY)   ;每路大小 = 1 << SWAY
NSET      EQU (CSIZE - NWAY - CLINE) ;每路包含的 cache 行数 = 1 << NSET

```

头文件中的值(value)或者是用以 2 为底的对数表示的 cache 的大小,或者表示域位置指针(field locator)。如果值表示的是一个位置指针(locator),那么它代表 CP15 寄存器位域的最低位。如本例中的常量 I7WAY 指向 CP15:c7:c5 寄存器的 way 选择域最低位。在 ARM920T, ARM922T, ARM940T 和 ARM1022E 中, I7WAY 的值为 26; 在 ARM926EJ-S, ARM946E-S 和 ARM1026EJ-S 中, I7WAY 的值为 30(见图 12.10)。数值以这种格式存储, 可以支持使用 MCR 指令发出清理命令时, 将内核寄存器(Rm9)数据向 CP15:Cd;Cm 寄存器搬移的位操作。

现将头文件中依赖于内核体系结构的 6 个常量分别列出:

- **CSIZE** 指 cache 全部字节数以 2 为基数取对数所得到的值。换句话说, cache 的容量就是 $(1 \ll CSIZE)$ 个字节。
- **CLINE** 指 cache 行中的字节数以 2 为基数取对数所得到的值。所以 cache 行的长度为 $(1 \ll CLINE)$ 个字节。
- **NWAY** 指路(way)的数量, 与组相联相同。

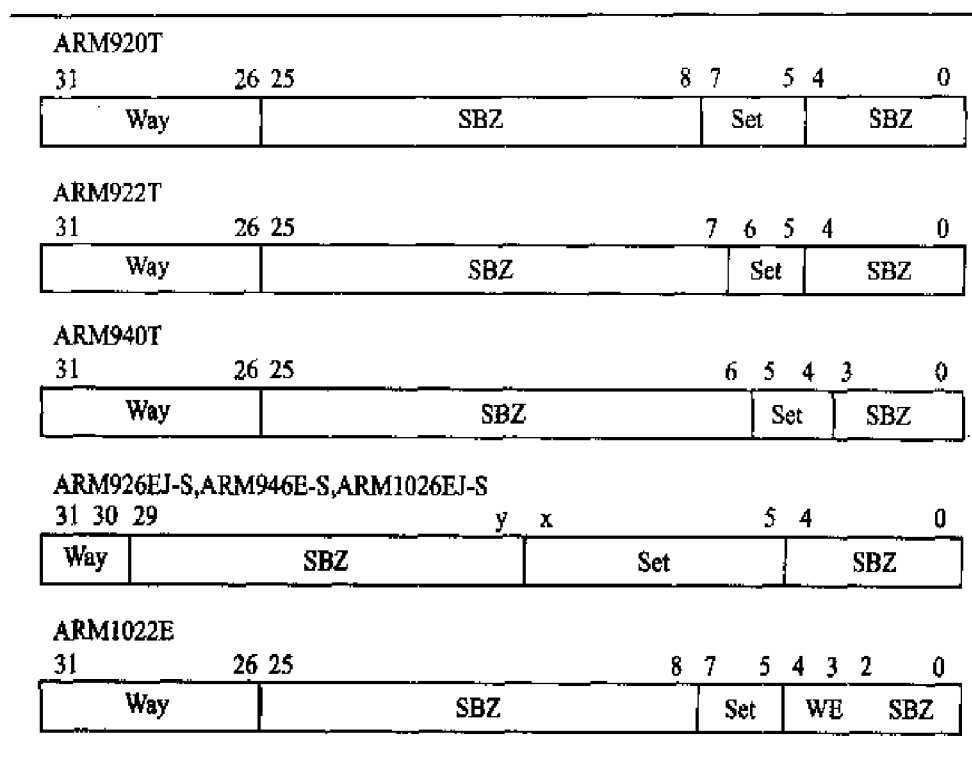


图 12.10 使用路和组索引寻址清理 cache 时 CP15:c7:Cm 寄存器 Rd 的格式

- **I7SET** 组索引在 CP15:c7 指令寄存器中左移的位数。这个值也被用作顺序访问 cache 时,在 CP15:c7 寄存器中增大或者减小组索引部分的大小。
- **I7WAY** 路索引在 CP15:c7 指令寄存器中左移的位数。这个值也被用作顺序访问 cache 时,在 CP15:c7 寄存器中增大或者减小路索引部分的大小。
- **I9WAY** 路索引在 CP15:c9 指令寄存器中左移的位数。这个值也被用作顺序访问 cache 时,在 CP15:c9 寄存器中增大或者减小路索引部分的大小。

还有 2 个由内核特殊数据计算而来的常量:

- **SWAY** 一路中的字节数以 2 为基数取对数所得到的值。所以一个 way 的大小就是 $(1 \ll \text{SWAY})$ 个字节。
- **NSET** 指每路中的 cache 行数,即组索引(set index)的长度以 2 为基数取对数所得到的值。组的数量为 $(1 \ll \text{NSET})$ 。

12.5.4 使用路和组索引寻址清理 D-cache

有些 ARM 内核支持用路(way)和组(set)索引寻址来确定某一 cache 行在 cache 中的位置,并清理和清除该单一的 cache 行。表 12.5 中以 MCR 指令的形式列出了清理和清除一个 cache 行的命令。有 2 个命令可以用来清除一个 cache 行:清除指令 cache 行和清除数

据 cache 行。其余的 2 个命令用来清理 D-cache:清理 cache 行和清理并清除 cache 行。

表 12.5 使用路和组索引寻址清理 cache 的 CP15:c7 寄存器命令

命 令	MCR 指令	所支持的内核
清除指令 cache 行	MCR p15,0,Rd,c7,c5,2	ARM926EJ-S,ARM940T,ARM1026EJ-S
清除数据 cache 行	MCR p15,0,Rd,c7,c6,2	ARM926EJ-S,ARM940T,ARM1026EJ-S
清理数据 cache 行	MCR p15, 0, Rd, c7, c10,2	ARM920T,ARM922T, ARM926EJ-S, ARM940T,ARM946E-S,ARM1022E, ARM1026EJ-S
清理并清除数据 cache 行	MCR p15, 0, Rd, c7, c14,2	ARM920T,ARM922T, ARM926EJ-S, ARM940T,ARM946E-S,ARM1022E, ARM1026EJ-S

列出的每个内核通过路和组索引寻址方式选定特定的某 cache 行。当使用这些指令时,在同样的 ARM 处理器内核上执行表 12.5 中的 4 条指令,内核寄存器 Rd 中的值是相同的;但是不同的处理器的寄存器位域格式是不同的。图 12.10 所示为支持通过路寻址方式清理和清除一个 cache 行的 ARM 内核的 CP15:c7:Cm 寄存器格式。在内核寄存器(Rd)中以相应的 CP15:c7 寄存器格式创建一个数值(value),就可以执行这些命令。寄存器通常包含两个位域(bit field):选择路和选择该路中的组。一旦寄存器被创建,执行相应的 MCR 指令就可以把内核寄存器(Rd)中的内容放到 CP15:c7 寄存器中。

在下面的例子中列出了在 ARM920T, ARM922T, ARM940T, ARM946E-S 和 ARM1022E 处理器上的 cleanDCache, cleanFlushDCache 和 cleanFlushCache 程序。

【例 12.3】 这里使用一个名为 CACHECLEANBYWAY 的宏创建使用路和组索引寻址方式清理、清除和清理清除 cache 的 3 个过程。

所定义的宏使用头文件 cache.h 中的常量以 CP15:c7 寄存器格式(c7f)为相应的处理器内核建立一个处理器寄存器。第一步将 c7f 寄存器清零,在 MCR 指令中,这被作为 Rd 输入值来执行相应的操作;接下来宏根据图 12.10 的格式,每次写 cache 行都会增加一次 c7f 寄存器。组索引在内循环中增加,路索引在外循环中增加。使用嵌套的循环,就可以逐步清理所有路当中的所有 cache 行。

```
AREA cleancachebyway, CODE, READONLY ;Start of Area block
```

```
IF {CPU} = "ARM920T"      :LOR; \
    {CPU} = "ARM922T"      :LOR; \
    {CPU} = "ARM940T"      :LOR; \
    {CPU} = "ARM946E-S"    :LOR; \
```

ARM 嵌入式系统开发

```
{CPU} = "ARM1022E"
```

```
EXPORT cleanDCache
```

```
EXPORT cleanFlushDCache
```

```
EXPORT cleanFlushCache
```

```
INCLUDE cache.h
```

```
c7f RN 0 ;cp15,c7 register format
```

```
MACRO
```

```
CACHECLEANBYWAY $op
```

```
MOV c7f, #0 ;建立 c7 格式
```

```
5
```

```
IF "$op" = "Dclean"
```

```
MCR p15, 0, c7f, c7, c10, 2 ;清理 D-cline
```

```
ENDIF
```

```
IF "$op" = "Dcleanflush"
```

```
MCR p15, 0, c7f, c7, c14, 2 ;清理、清除 D-cline
```

```
ENDIF
```

```
ADD c7f, c7f, #1 << I7SET ;组索引 + 1
```

```
TST c7f, #1 << (NSET + I7SET) ;测试索引溢出
```

```
BEQ %BT5
```

```
BIC c7f, c7f, #1 << (NSET + I7SET) ;清空索引溢出
```

```
ADDS c7f, c7f, #1 << I7WAY ;丢弃者指针 + 1
```

```
BCC %BT5 ;测试路溢出
```

```
MEND
```

```
cleanDCache
```

```
CACHECLEANBYWAY Dclean
```

```
MOV pc, lr
```

```
cleanFlushDCache
```

```
CACHECLEANBYWAY Dcleanflush
```

```
MOV pc, lr
```

```
cleanFlushCache
```

```
CACHECLEANBYWAY Dcleanflush
```

```

MCR      p15,0,r0,c7,c5,0      ;清除 I-cache
MOV      pc, lr
ENDIF

```

12.5.5 使用 test-clean 命令清理 D-cache

2 种较新的 ARM 内核 ARM926EJ - S 和 ARM1026EJ - S 可以使用 test-clean(测试清理) CP15;c7 寄存器清理 cache 行。test-clean 命令是一条特殊的清理指令,它用在软件循环中可以非常有效地清理 cache。ARM926EJ - S 和 ARM1026EJ - S 同样也支持使用组索引和路索引来清理 cache,但是使用 test-clean 命令清理 cache 可以更加高效。

在下面的程序中,使用表 12.6 中所示的命令清理 ARM926EJ - S 和 ARM1026EJ - S 内核。

表 12.6 测试清理单一 D-cache 行的命令

命 令	MCR 指令	支持的内核
通过循环测试清理 D-cache 行	MCR p15,0,r15,c7,c10,3	ARM926EJ - S, ARM1026EJ - S
通过循环测试清理并清除 D-cache	MCR p15,0,r15,c7,c14,3	ARM926EJ - S, ARM1026EJ - S

【例 12.4】 ARM926EJ - S 和 ARM1026EJ - S 内核的 cleanDCache, cleanFlushD-Cache 和 cleanFlushCache 程序。

测试清理命令找到第一个带有脏位的 cache 行,将其中的内容传送到主存中,从而清理 cache。如果在 cache 中还有其它的脏位,那么 Z 标志位就会被清零。

```
IF {CPU} = "ARM926EJ - S" ;LOR; {CPU} = "ARM1026EJ - S"
```

```

EXPORT cleanDCache
EXPORT cleanFlushDCache
EXPORT cleanFlushCache

```

cleanDCache

```

MRC      p15, 0, pc, c7, c10, 3      ;测试清理 D-cline
BNE      cleanDCache
MOV      pc, lr

```

cleanFlushDCache

```

MRC      p15, 0, pc, c7, c14, 3      ;测试清理并清除 D-cline
BNE      cleanFlushDCache

```

```

        MOV     pc, lr
cleanFlushCache
        MRC     p15, 0, pc, c7, c14, 3      ;测试清理并清除 D-cline
        BNE     cleanFlushCache
        MCR     p15, 0, r0, c7, c5, 0       ;清除 I-cache
        MOV     pc, lr
    ENDIF

```

清理 cache 可以通过创建一个使用 test-clean 命令的软件循环来实现。通过测试 Z 标志位并跳转重复测试,处理器循环测试直到 D-cache 被清理。需要注意的是, test-clean 命令使用程序计数器(r15)作为 Rd 寄存器到 MCR 指令的输入。

12.5.6 在 Intel XScale SA - 110 和 Intel StrongARM 内核中清理 D-cache

Intel XScale 和 Intel StrongARM 处理器使用第 3 种方法来清理其 D-cache。Intel XScale 处理器有一种命令可以在 D-cache 中分配一行而不需要填充它。当处理器执行这条命令时,有效位(valid bit)被置 1,并使用 Rd 寄存器提供的 cache 标签(cache-tag)填充目录项(directory entry)。并且,当这条命令被执行时,主存不会向 cache 传送数据。这样,cache 中的数据直到被处理器写操作时才会被初始化。表 12.7 中的分配命令有自动替换出脏 cache 行的优点。

表 12.7 Intel Xscale 分配 D-cache 行的 CP15:c7 命令

命 令	MCR 指令	支持的内核
在数据 cache 中分配行	MCR p15,0,Rd,c7,c2,5	XScale

Intel StrongARM 和 Intel Xscale 处理器需要附加的技术来清理 cache。它们需要一块专用的、未被使用的、且可 cache 的主存来清理 cache。通过相应的软件设计,可以使这个存储器块专门用作清理 cache。

由于使用轮转替换策略,Intel StrongARM 和 Intel Xscale 处理器可以通过读固定的主存块清理 cache。如果一个程序的执行使处理器内核顺序读主存中与 cache 大小相同的一块,那么这一系列的读操作将会把所有当前 cache 行替换出 cache,并把读取的数据块放到 cache 中。当顺序读操作结束时,cache 中不会包含任何重要的数据,因为专用的读块中没有有用的信息。所以 cache 可以被清除而不用担心丢失有用信息。

可使用这种技术来清理 Intel StrongARM 的 D-cache 和 Intel Xscale 的迷你 D-cache。Intel StrongARM 和 Intel Xscale 处理器的 cleanDCache, cleanFlushDCache 和 cleanFlush-

Cache 过程的源程序在下面的例子中。CleanMiniDCache 是一个附加程序,用来清理 Intel XScale 处理器的 D-cache。

【例 12.5】 本例使用了 2 个宏:CPWAIT 和 CACHECLEANXSCALE。CPWAIT 是一个 3 指令序列,用以在 XScale 处理器上防止 CP15 操作的副作用。由于宏使用这些指令,所以有足够的处理器周期执行,从而确保 CP15 命令的完成并且流水线中已经没有指令。CPWAIT 宏定义如下:

```
MACRO
CPWAIT
MRC      p15,0,r12,c2,c0,0      ;读所有 CP15 寄存器
MOV      r12,r12;
SUB      pc,pc,#4;              ;跳到下一条指令
MEND
```

宏 CACHECLEANXSCALE 创建了 cleanDCache, cleanFlushDCache 和 cleanFlushCache 3 个过程。宏的第 1 部分为程序设定了物理参数。第 1 个参数 adr 是用来清理 cache 的专用存储块的虚拟起始地址。第 2 个参数 nl 是 cache 中 cache 行的总数。

```
IF {CPU} = "XSCALE";LOR; {CPU} = "SA-110"
    EXPORT cleanDCache
    EXPORT cleanFlushDCache
    EXPORT cleanFlushCache
    INCLUDE cache.h
```

```
CleanAddressDcache      EQU 0x8000;  (32k block  0x8000 - 0x10000)
CleanAddressMiniDcache  EQU 0x10000; (2k block   0x10000 - 0x10800)
```

```
adr      RN  0      ;起始地址
nl       RN  1      ;待处理的 cache 行数
tmp      RN  12     ;临时寄存器
```

```
MACRO
CACHECLEANXSCALE $ op

IF "$ op" = "Dclean"
    LDR      adr, = CleanAddressDcache
    MOV      nl, #(1 << (NWAY + NSET))
ENDIF
```

ARM 嵌入式系统开发

```

        IF "$ op" = "DcleanMini"
            LDR        adr, = CleanAddressMiniDcache
            MOV        nl, #(1 << (MNWAY + NSET))
        ENDIF

5
        IF {CPU} = "XSCALE";LAND; "$ op" = "Dclean"
            MCR        p15, 0, adr, c7, c2, 5      ;分配 D-cache 行
            ADD        adr, adr, #32                ;+1 D-c line
        ENDIF
        IF {CPU} = "SA-110";LOR; "$ op" = "DcleanMini"
            LDR        tmp,[adr],#32                ;装载数据, D-cache 行+1
        ENDIF
        SUBS        nl, nl, #1                      ; - 1 loop count
        BNE        %BTS
        IF {CPU} = "XSCALE"
            CPWAIT
        ENDIF
        MEND

cleanDCache
        CACHECLEANKSCALE Dclean
        MOV pc, lr

cleanFlushDCache
        STMFID       sp!, {lr}
        BL cleanDCache
        IF {CPU} = "XSCALE"
            BL cleanMiniDCache
        ENDIF
        MOV        r0, #0
        MCR        p15,0,r0,c7,c6,0                ;清除 D-cache
        IF {CPU} = "XSCALE"
            CPWAIT
        ENDIF
        LDMFID       sp!, {pc}

cleanFlushCache
        STMFID       sp!, {lr}
        BL cleanDCache

```

```

IF {CPU} = "XSCALE"
    BL cleanMiniDCache
ENDIF

MOV      r0, #0
MCR      p15,0,r0,c7,c7,0      ;清除 I-cache 和 D-cache
IF {CPU} = "XSCALE"
    CPWAIT
ENDIF

LDMFD    sp!, {pc}
ENDLF

IF {CPU} = "XSCALE"
    EXPORT cleanMiniDCache

cleanMiniDCache
    CACHECLEANXSCALE DcleanMini
    MOV pc, lr
ENDLF

```

接下来,宏会筛选所需要的命令来执行 2 个处理器内核的清理操作。Intel XScale 使用 CP15;c7 行分配命令来清理 D-cache,并通过读取专用的存储器块来清理迷你 D-cache。Intel StrongARM 读取存储器中特定的一块来清理其 D-cache。

最后,可多次使用宏来创建 cleanDCache, cleanFlushDCache, cleanFlushCache 和 clean-MiniDCache 等过程。

421

12.5.7 清理和清除部分 cache

ARM 内核支持通过访问主存中某一个 cache 行所对应的主存地址来清理和清除该 cache 行。表 12.8 以 MCR 指令的形式列出了这些命令。这些命令中的 2 个可以用来清除单一的 cache 行:一个命令负责清除指令 cache;另一个命令清除数据 cache。另外 2 个命令用来清理数据 cache;一个清理单一的 cache 行;另一个清理并清除一个 cache 行。

表 12.8 通过主存中所对应的地址清理和清除一个 cache 行的命令

命 令	MCR 指令	支持的内核
清除指令 cache 行	MCR p15,0,Rd,c7,c5,1	ARM920T, ARM922T, ARM926EJ-S, ARM946E-S, ARM1022E, ARM1026EJ-S, XScale
清除数据 cache 行	MCR p15,0,Rd,c7,c6,1	ARM920T, ARM922T, ARM926EJ-S, ARM946E-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale
清理数据 cache 行	MCR p15,0,Rd,c7,c10,1	ARM920T, ARM922T, ARM926EJ-S, ARM946E-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale
清理并清除数据 cache 行	MCR p15,0,Rd,c7,c14,1	ARM920T, ARM922T, ARM926EJ-S, ARM946E-S, ARM1022E, ARM1026EJ-S, XScale

当使用这些指令时,在同一个处理器上,内核寄存器 Rd 的值对于这 4 个命令是相同的,并且它的内容必须能够将 CP15;c7 寄存器置位;然而,对于不同的处理器,CP15;c7 寄存器位值(bit value)的格式稍有不同。图 12.11 所示为支持清理和清除一个 cache 行的内核的寄存器格式。如果内核支持 MMU,那么清理和清除一个 cache 行可以通过改变虚拟地址来实现;如果内核支持 MPU,那么可以通过改变物理地址实现。

这里使用这 4 个命令来创建 6 个例程,用来清理、清除或者既清理又清除 cache 中代表主存中某个区域的 cache 行。

- flushICacheRegion 清除 I-cache 中代表主存中一块区域的 cache 行;
- flushDCacheRegion 清除 D-cache 中代表主存中一块区域的 cache 行;
- cleanDCacheRegion 清理 D-cache 中代表主存中一块区域的 cache 行;
- cleanFlushDCacheRegion 清理并清除 D-cache 中代表主存中一块区域的 cache 行;
- flushCacheRegion 清除 D-cache 和 I-cache 中代表主存中一块区域的 cache 行;
- cleanFlushCacheRegion 清理并清除 D-cache,接下来清除 I-cache。

对于所有的过程都有 2 个输入参数:主存中的起始地址(adr)和区域中的字节数(b)。C 函数原型如下:

```
void flushICacheRegion(int * adr,unsigned int b);  
void flushDCacheRegion(int * adr,unsigned int b);
```

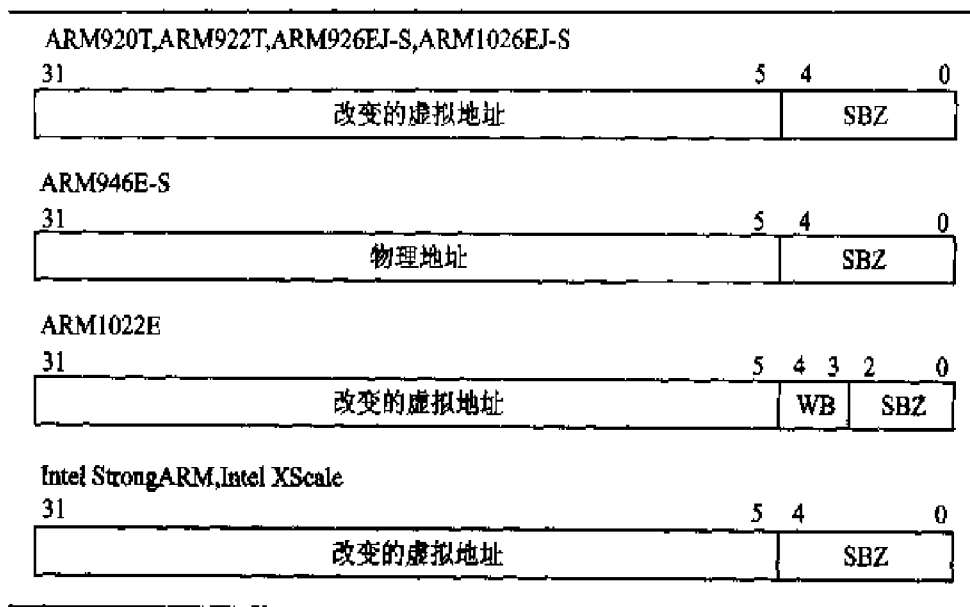


图 12.11 通过主存中所对应的地址清理和清除 cache 行时 CP15:c7 寄存器的格式

```
void cleanDCacheRegion(int * adr,unsigned int b);
void cleanFlushDCacheRegion(int * adr,unsigned int b);

void flushCacheRegion(int * adr,unsigned int b);
void cleanFlushCacheRegion(int * adr,unsigned int b);
```

当使用以上指令清理 cache 区域过程时要小心,它们比较适合小存储区域。如果区域大小比 cache 大好几倍,那么使用 12.5.4,12.5.5 和 12.5.6 小节中介绍的清理 cache 过程来清理整个 cache,反而是更加有效的方法。

只有少数 ARM 内核支持区域过程。同样这些内核也列在接下来的示例代码的开始部分中。

【例 12.6】 宏将输入的地址裁剪成一个 cache 行大小,这种裁剪通常将地址对应到 ARM1022E 内核的 cache 行的第一个双字(double word)(见图 12.11)。然后,宏会将输入参数字节数 *b* 转换成 cache 行数。宏将 cache 行的数目作为计数器变量在选中的清除和清理操作中作循环,每一次循环结束后,将地址增加一个 cache 行的大小,直到计数器计数值为零。

```
IF {CPU} = "ARM920T" ;LOR; \
{CPU} = "ARM922T" ;LOR; \
{CPU} = "ARM946E-S" ;LOR; \
{CPU} = "ARM926EJ-S" ;LOR; \
```

```

{CPU} = "ARM1022E"      ;LOR: \
{CPU} = "ARM1026EJ-S"   ;LOR: \
{CPU} = "XSCALE"        ;LOR: \
{CPU} = "SA-110"

```

```

INCLUDE cache.h

```

```

adr      RN  0      ;active address
size     RN  1      ;size of region in bytes
nl       RN  1      ;要清理或者清除的 cache 行数

```

```

MACRO

```

```

CACHEBYREGION $op

```

```

BIC      adr, adr, #(1 << CLINE) - 1      ;clip 2 cline adr
MOV      nl, size, lsr #CLINE              ;bytes to cline

```

```

10

```

```

IF "$op" = "IcacheFlush"

```

```

    MCR    p15, 0, adr, c7, c5, 1          ;清除 I-cline@adr

```

```

ENDIF

```

```

IF "$op" = "DcacheFlush"

```

```

    MCR    p15, 0, adr, c7, c6, 1          ;清除 D-cline@adr

```

```

ENDIF

```

```

IF "$op" = "IDcacheFlush"

```

```

    MCR    p15, 0, adr, c7, c5, 1          ;清除 I-cline@adr

```

```

    MCR    p15, 0, adr, c7, c6, 1          ;清除 D-cline@adr

```

```

ENDIF

```

```

IF "$op" = "DcacheClean"

```

```

    MCR    p15, 0, adr, c7, c10, 1         ;清理 D-cline@adr

```

```

ENDIF

```

```

IF "$op" = "DcacheCleanFlush"

```

```

    IF {CPU} = "XSCALE" ;LOR;\

```

```

        {CPU} = "SA-110"

```

```

        MCR p15, 0, adr, c7, c10, 1        ;清理 D-cline@adr

```

```

        MCR p15, 0, adr, c7, c6, 1         ;清除 D-cline@adr

```

```

    ELSE

```

```

        MCR p15, 0, adr, c7, c14, 1        ;清理并清除 D-cline@adr

```

```

    ENDIF

```

```

ENDIF
IF "$ op" = "IDcacheCleanFlush"
    IF {CPU} = "ARM920T"      ;LOR; \
        {CPU} = "ARM946E-S"  ;LOR; \
        {CPU} = "ARM1022E"   ;LOR; \
        {CPU} = "ARM922T"    ;LOR; \
        {CPU} = "ARM926EJ-S" ;LOR; \
        {CPU} = "ARM1026EJ-S"
        MCR p15, 0, adr, c7, c14, 1      ;清理并清除 D-cline@adr
        MCR p15, 0, adr, c7, c5, 1       ;清除 I-cline@adr
    ENDIF
    IF {CPU} = "XSCALE"
        MCR p15, 0, adr, c7, c10, 1      ;清理 D-cline@adr
        MCR p15, 0, adr, c7, c6, 1       ;清除 D-cline@adr
        MCR p15, 0, adr, c7, c5, 1       ;清除 I-cline@adr
    ENDIF
ENDIF

ADD      adr, adr, #1 << CLINE          ; + 1 next cline adr
SUBS     nl, nl, #1                     ; - 1 cline counter
BNE      %BT10                          ;清除 # lines + 1
IF {CPU} = "XSCALE"
    CPWAIT
ENDIF
MOV      pc, lr
MEND

IF {CPU} = "SA-110"
    EXPORT cleanDcacheRegion
    EXPORT flushDcacheRegion
    EXPORT cleanFlushDcacheRegion
cleanDcacheRegion
    CACHEBYREGION DcacheClean
flushDcacheRegion
    CACHEBYREGION DcacheFlush
cleanFlushDcacheRegion
    CACHEBYREGION DcacheCleanFlush

```

```

        ELSE
            EXPORT flushICacheRegion
            EXPORT flushDCacheRegion
            EXPORT flushCacheRegion
            EXPORT cleanDCacheRegion
            EXPORT cleanFlushDCacheRegion
            EXPORT cleanFlushCacheRegion

flushICacheRegion
        CACHEBYREGION IcacheFlush

flushDCacheRegion
        CACHEBYREGION DcacheFlush

flushCacheRegion
        CACHEBYREGION IDcacheFlush

cleanDCacheRegion
        CACHEBYREGION DcacheClean

cleanFlushDCacheRegion
        CACHEBYREGION DcacheCleanFlush

cleanFlushCacheRegion
        CACHEBYREGION IDcacheCleanFlush

    ENDIF
ENDIF

```

最后,使用名为 CACHEBYREGION 的宏创建例程。对于命令集有限的 Intel Strong-ARM 内核,可以创建 3 个例程(procedure);对于其余的拥有分离 cache 的处理器,可以创建所有的 6 个例程。

12.6 cache 锁定

cache 锁定(lockdown)是 cache 的一项特性,使程序能够加载对时间要求很苛刻的代码和数据到 cache 中来,并将这些代码和数据标记为*非替换(exempt of eviction)*的。被锁定的代码和数据有更快的系统反映能力,因为这些数据和代码一直存放在 cache 中。cache 在正常操作时,经常会涉及到 cache 行替换,这种替换会带来代码执行时间不确定的问题,而 cache 锁定可以避免这种不确定性。

将信息锁定在 cache 中的目的是,避免 cache 失效所造成的负面效果。然而,由于任何用作 cache 锁定的 cache 存储器单元不能够再存储主存的其它内容,所以,可用的 cache 空

间被减少了。

ARM 内核为 cache 锁定分配固定的 cache 单元。一般来讲,分配作 cache 锁定的 cache 单元是一个路(way)。例如:一个 4 路组相联 cache 允许将锁定的代码和数据放在容量为 cache 总容量的 1/4 的 cache 单元内。带 cache 的内核通常至少保留一个路作为 cache 的正常操作使用。

有些指令往往需要被锁定在 cache 中,如中断向量表、中断服务程序以及为一些特殊算法编制的代码。这些算法被系统广泛使用且时间要求比较苛刻。对于数据来说,经常使用到的全局变量最好被锁定在 cache 中。

锁定在 ARM cache 中的数据和代码不会被替换。但是,如果 cache 被清除,被锁定的信息也会丢失,存放锁定信息的 cache 存储区也不能被用作一般的 cache 存储区。必须重新运行 cache 锁定程序,以保存新的锁定信息。

12.6.1 在 cache 中锁定代码和数据

本小节将介绍如何在 cache 中锁定代码和数据。锁定代码和数据的典型 C 程序如下:

```
int interrupt_state;           /* 保存 FIQ 和 IRQ 位的状态 */
int globalData[16];
unsigned int * vectortable = ( unsigned int * ) 0x0;
int wayIndex;
int vectorCodeSize = 212;      /* 向量表和 FIQ 句柄的字节数 */

interrupt_state = disable_interrupts(); /* 不提供代码 */
enableCache();                 /* 详见第 13 章(MPU)和第 14 章(MMU) */
flushCache();                  /* 见例 12.2 */

/* 锁定全局数据块 */
wayIndex = lockDcache( globalData, sizeof globalData);
/* 锁定中断向量表和 FIQ 句柄 */
wayIndex = lockIcache( vectortable, vectorCodeSize);

enable_interrupts(interrupt_state); /* 不提供代码 */
```

开始时,中断被禁止,而 cache 是使能的。禁止中断的程序在这里没有表述。其中 flushCache 程序详见本章前几节的例子。实际使用的调用取决于 cache 的配置,而且很可能也包括 cache 的清理。

函数 lockDcache 在 D-cache 中锁定一块数据;类似的,函数 lockIcache 在 I-cache 中锁

ARM 嵌入式系统开发

定一个代码块。

执行 cache 锁定的软件本身必须被存放在不可 cache 的(noncached)主存中。锁定在 cache 中的代码和数据必须被存放在可 cache 的(cached)主存中。被锁定在 cache 中的代码和数据不能存在于 cache 中的其它地方,这一点非常重要。换句话说,如果 cache 中的内容未知,那么在装载之前应先清除 cache。如果内核使用的是回写 D-cache,那么应清理 D-cache。一旦代码和数据被装载到 cache,就可以重新使能中断了。

对于函数 lockDcache 和函数 lockIcache,这里提供了 3 种不同的实现代码,因为根据体系结构方式的不同,在 cache 中锁定代码有 3 种不同的方法。第 1 种锁定代码和数据的方法使用了路(way)寻址技术;第 2 种使用了一组锁定位(lock bit);在第 3 种方法中,结合使用了特殊分配命令和读取主存中特定块这 2 种方法,来锁定代码和数据。

表 12.9 列出了实现 lockDcache 和 lockIcache 的 3 个例子、使用的方法以及相关的处理器。

表 12.9 cache 锁定的方法

示 例	过程方法	处理器内核
例 12.7	路(way)寻址	ARM920T, ARM926EJ-S, ARM940T, ARM946E-S, ARM1022E, ARM1026EJ-S
例 12.8	锁定位(lock bits)	ARM926EJ-S, ARM1026EJ-S
例 12.9	特殊分配命令	Xscale

12.6.2 通过增加路索引来锁定 cache

ARM920T, ARM926EJ-S, ARM940T, ARM946E-S, ARM1022E 和 ARM1026EJ-S 使用路和组索引寻址来实现锁定。2 个 CP15:c9:c0 寄存器中包含 12.3.2 小节中描述的丢弃者计数器的复位寄存器(reset registers)。这 2 个寄存器中的一个控制着 I-cache,另一个控制 D-cache。这些寄存器被用作在一个路中选择用来锁定数据和代码的 cache 行。

写到 CP15:c7 寄存器中的值用来设置丢弃计数复位值(victim reset value),即当丢弃者计数器的值增加到超出内核中路数目时,丢弃者计数器被复位的值。系统上电时,复位值为 0,只有当 cache 中的某一部分被用作锁定时,复位值才由软件改变。当 cache 中的某一部分被用作锁定时,可用 cache 行的数量随被锁定的 cache 行数的增加而减少。读此寄存器,将返回当前的丢弃计数复位值。读和写这 2 个寄存器的 MRC 和 MCR 指令见表 12.10。

表 12.10 通过访问路,在 cache 中锁定数据的命令

命 令	MRC 和 MCR 指令	处理器内核
读 D-cache 锁定基	MRC p15,0,Rd,c9,c0,0	ARM920T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S
写 D-cache 锁定基	MCR p15,0,Rd,c9,c0,0	ARM920T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S
读 I-cache 锁定基	MRC p15,0,Rd,c9,c0,1	ARM920T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S
写 I-cache 锁定基	MCR p15,0,Rd,c9,c0,1	ARM920T,ARM926EJ-S,ARM940T, ARM946E-S,ARM1022E,ARM1026EJ-S

当读/写锁定基地址时,对于不同的处理器来说,MRC 和 MCR 指令使用的内核寄存器 Rd 的格式稍有不同。图 12.12 所示为使用这些指令的处理器的内核 Rd 寄存器格式。为保证命令正确地执行,一定要使 Rd 寄存器的格式与图 12.12 所示一致。

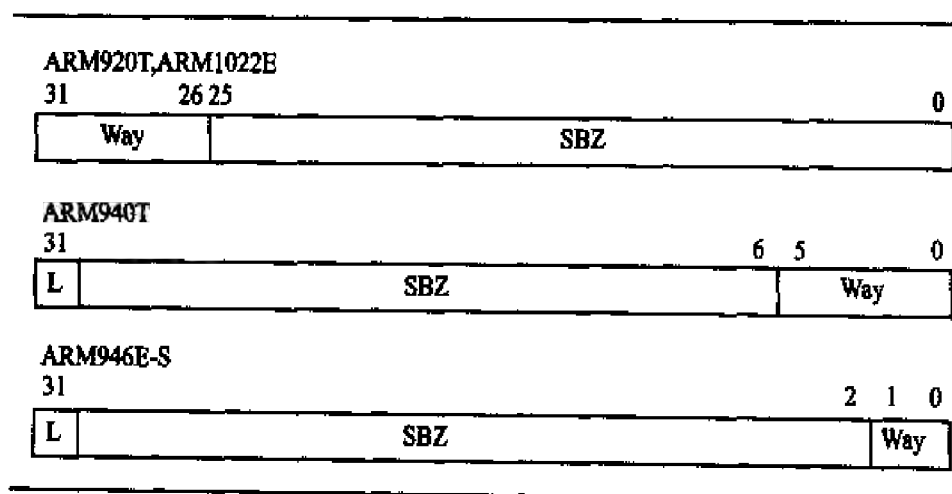


图 12.12 通过访问路,在 cache 中锁定数据时 CP15:c9 寄存器格式

将指令锁定在 cache 中还需要一条特殊的装载命令。该命令复制与 cache 行相同大小的主存块到 I-cache 的 cache 行中。该命令以及 Rd 寄存器在该指令中使用的格式见表 12.11 和图 12.13。

表 12.11 在 I-cache 中锁定 cache 行

命令	MCR 指令	处理器内核
通过地址预取 I-cache 行	MCR p15,0,Rd,c7,c13,1	ARM920T, ARM922T, ARM926EJ-S, ARM940T, ARM946E-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale

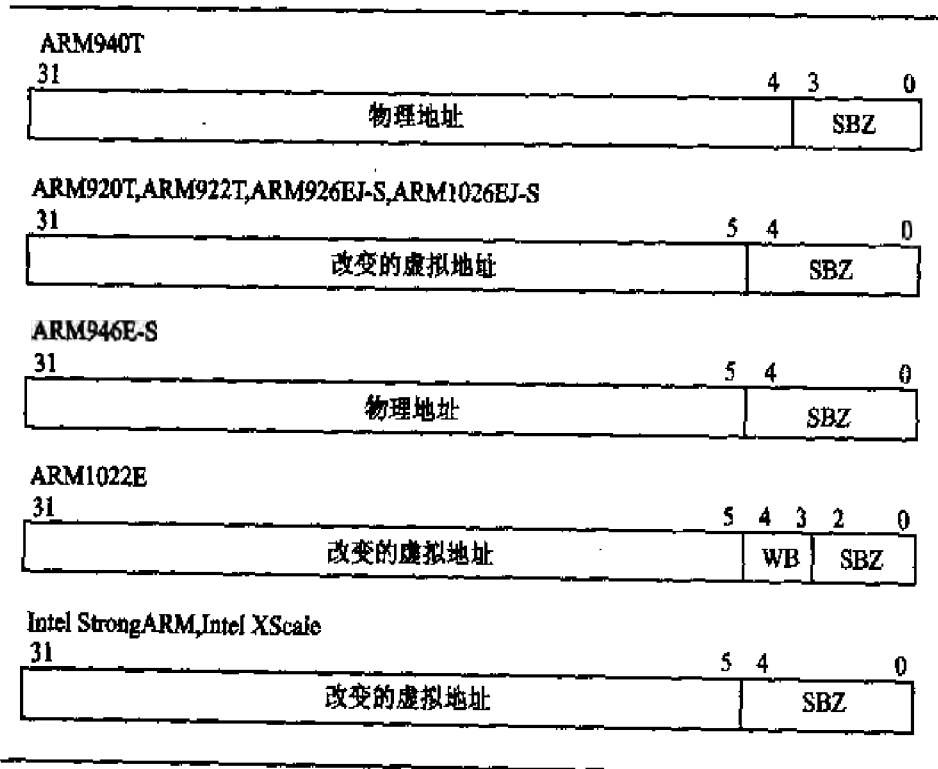


图 12.13 在 I-cache 中锁定 cache 行时 CP15:c7:c13

【例 12.7】这个例程的第一部分定义了宏 CACHELOCKBYWAY 中使用的寄存器。在宏的定义中也使用了头文件 cache.h 中的常量。

宏的第一行将地址(adr)调整成 cache 行地址。接下来的 3 行根据代码的字节数决定承载代码所需要的路数。然后由 CP15:c9:c0 中读取 I-cache 和 D-cache 的当前被丢弃者指针。

接下来的几行代码做错误检测,主要测试 cache 是否过载(overflowing)以及即将装载的代码长度是否为零。

在 ARM940T 和 ARM946E-S 中锁定代码和数据时,在将一块内存数据锁定在 cache 行中以前,必须把锁定位(lock bit)置位。本例程中的下一条指令就是将锁定位置位,并将数据写回到 CP15:c9:c0 寄存器。

在此,代码使用了嵌套的循环:外面的大循环选择路,里面的小循环在一个路中不断增

加 cache 行地址。

在这 2 个循环的中央,使用预取指令或者装载数据命令在 cache 存储器中锁定一个 cache 行。为了锁定指令,宏对一个特殊的 CP15:c7:c13 寄存器执行写入操作,从主存中预装载代码段。为了锁定数据,只须使用 LDR 指令读数据即可。

如果内核为 ARM940T 或 ARM946E-S,那么宏退出返回时须清除 CP15:c9:c0 寄存器中的锁定位。对于所有的 ARM 内核,宏在锁定的代码和数据之后将丢弃者指针指向下一个路。

```
IF {CPU} = "ARM920T"      ;LOR; \
    {CPU} = "ARM922T"      ;LOR; \
    {CPU} = "ARM940T"      ;LOR; \
    {CPU} = "ARM946E-S"    ;LOR; \
    {CPU} = "ARM1022E"
EXPORT lockDCache
EXPORT lockICache
INCLUDE cache.h
```

adr	RN 0	;代码和数据的当前地址
size	RN 1	;cache 存储器的字节数
nw	RN 1	;存储器中的路数
count	RN 2	
tmp	RN 2	;临时寄存器
tmp1	RN 3	;临时寄存器
c9f	RN 12	;CP15:c9 寄存器格式

MACRO

CACHELOCKBYWAY \$ op

BIC	adr, adr, #(1 << CLINE) - 1	;调整地址为 cache 行地址
LDR	tmp, = (1 << SWAY) - 1	;scratch = 路的大小
TST	size, tmp	;路的末尾有没有碎片?
MOV	nw, size, lsr #SWAY	;将字节数转换成路数
ADDNE	nw, nw, #1	;如果有碎片,则路数加 1
CMP	nw, #0	;没有锁定请求
BEQ	%FT2	;退出并返回丢弃者基

IF "\$ op" = "Icache"

ARM 嵌入式系统开发

```

        MRC          p15, 0, c9f, c9, c0, 1          ;获取 I-cache 的丢弃者
    ENDIF
    IF "$ op" = "Dcache"
        MRC          p15, 0, c9f, c9, c0, 0          ;获取 D-cache 的丢弃者
    ENDIF

    AND              c9f, c9f, tmp                    ;屏蔽高位 c9f = victim
    ADD              tmp, c9f, nw                      ;temp = 丢弃者 + 路计数值
    CMP              tmp, #(1 << NWAY) - 1            ;> 总的路数 ?
    MOVGT            r0, #-1                          ;如果路数过多,则返回 -1
    BGT              %FT1                             ;错误: cache 路溢出

    IF {CPU} = "ARM940T" ;LDR: {CPU} = "ARM946E-S"
        ORR          c9f, c9f, #1 << 31             ;将 cache 设为锁定模式
    ENDIF

10
    IF "$ op" = "Icache"
        MCR          p15, 0, c9f, c9, c0, 1          ;设置丢弃者
    ENDIF
    IF "$ op" = "Dcache"
        MCR          p15, 0, c9f, c9, c0, 0          ;设置丢弃者
    ENDIF

5
    MOV              count, #(1 << NSET) - 1

    IF "$ op" = "Icache"
        MCR          p15, 0, adr, c7, c13, 1          ;装载指令 cache 行
        ADD          adr, adr, #1 << CLINE            ;cache 行地址加 1
    ENDIF
    IF "$ op" = "Dcache"
        LDR          tmp1, [adr], #1 << CLINE        ;装载数据 cache 行
    ENDIF

    SUBS             count, count, #1
    BNE              %BT5
    ADD              c9f, c9f, #1 << I9WAY            ;丢弃者指针加 1
    SUBS             nw, nw, #1                      ;路计数器减 1

```

```

2
    BNE      %BT10      ;重复路的数目

    IF {CPU} = "ARM940T" ;LOR: {CPU} = "ARM946E-S"
        BIC      r0, c9f, #1 << 31      ;清除锁定位,并将丢弃者数值赋值给 r0
    ENDIF
    IF "$ op" = "Icache"
        MCR      p15, 0, r0, c9, c0, 1      ;设置丢弃者计数器
    ENDIF
    IF "$ op" = "Dcache"
        MCR      p15, 0, r0, c9, c0, 0      ;设置丢弃者计数器
    ENDIF

1
    MOV      pc, lr
    MEND

lockDCache
    CACHELOCKBYWAY Dcache
lockICache
    CACHELOCKBYWAY Icache
    ENDIF
```

最后,2次使用宏,创建 lockDCache 和 lockICache 函数。

12.6.3 使用锁定位锁定 cache

ARM926EJ-S 和 ARM1026EJ-S 使用一组锁定位在 cache 中锁定代码和数据,如图 12.14所示。这两种处理器的 CP15:c9 指令使用不同的 Rd 格式,见表 12.12。0~3 四个位(bit)分别代表了在两种处理器的 4 路组相联 cache 中的 4 个路。如果某位被置位,那么它所对应的路就被锁定。对于 D-cache,锁定了数据;对于 I-cache,则锁定了代码。被锁定的路中的 cache 行,直到被解锁之后才能够被替换。将 L 位中的某一位清零,就可以解锁相应的路。这种锁定 cache 的方式使系统代码可以单独选择锁定和解锁的路。

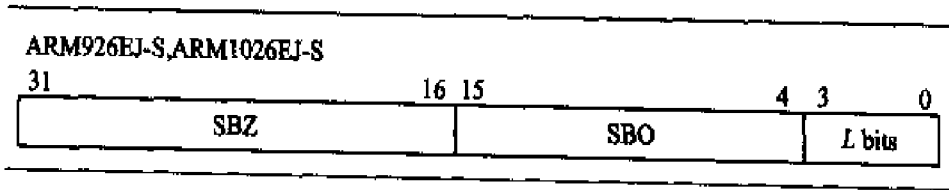


图 12.14 使用锁定位锁定 cache 时的 CP15:c9 寄存器格式

表 12.12 使用锁定位锁定 cache 的 CP15:c9 命令

命 令	MRC 和 MCR 指令
读 I-cache 锁定页寄存器	MRC p15,0, Rd,c9,c0,1
读 D-cache 锁定页寄存器	MRC p15,0, Rd,c9,c0,0
写 I-cache 锁定页寄存器	MCR p15,0, Rd,c9,c0,1
写 D-cache 锁定页寄存器	MCR p15,0, Rd,c9,c0,0
装载某一地址的代码 cache 行	MCR p15,0, Rd,c7,c13,1

这种单独选择锁定路的功能,使系统中代码的锁定和解锁更加容易。本小节中的示例程序使用了与其它带 cache 的内核相同的锁定数据的程序接口。

ARM926EJ-S 和 ARM1026EJ-S 的 lockDCache 和 lockICache 示例程序有相同的输入参数。但是代码的大小受路的最大尺寸的限制,并且可以重复调用 3 次。在本小节的例子中,锁定位 3 通常为 cache 专用的。这不是处理器硬件的限制,而仅仅是为了适应程序接口的需要,对过程调用做出的限制。

如果 size 参数是 1 字节或者更大,那么示例程序返回被锁定路的锁定位(L 位)。如果 size 参数为 0,那么程序返回下一个可用的路的锁定位。如果没有可锁定的路,那么将返回 8。

【例 12.8】 通过名为 CACHELOCKBYBIT 的宏产生 lockDCache 和 lockICache 函数。宏 CACHELOCKBYBIT 也使用了 cache.h 头文件所定义的常数。

宏首先检查 cache 中将要被锁定的字节数是否为零;接下来它分析承载这些代码所需要的 cache 行数,并调整地址 adr 到 cache 行地址。

如果程序要在 D-cache 中锁定数据,那么它就会读锁定寄存器 CP15:c9:c0:0 的值;如果要在 I-cache 中锁定代码,那么就会读锁定寄存器 CP15:c9:c0:1 的值。结果被放在内核 c9f 寄存器中,锁定位也被存放在 tmp 寄存器中,以备以后使用。

```

IF {CPU} = "ARM926EJ-S" ;LOR; \
{CPU} = "ARM1026EJ-S"
EXPORT lockDCache
EXPORT lockICache
EXPORT bittest
INCLUDE cache.h

adr      RN 0           ;代码和数据的当前地址
size     RN 1           ;存储器的字节数
tmp      RN 2           ;scratch 寄存器

```



```

tmp1    RN 3                                ;scratch 寄存器
c9f     RN 12                              ;CP15:c9 寄存器的格式

MACRO
CACHELOCKBYLBIT $op

    ADD     size, adr, size                ;size = 末地址(当前地址 + 存储器字节数)
    BIC     adr, adr, #(1 << CLINE) - 1    ;与 CLINE 对齐
    MOV     tmp, #(1 << CLINE) - 1        ;临时 CLINE 屏蔽
    TST     size, tmp                     ;CLINE 末尾有碎片?
    SUB     size, size, adr               ;补上对齐字节数
    MOV     size, size, lsr #CLINE        ;转换大小 2 # CLINE
    ADDNE   size, size, #1                ;为碎片加 CLINE
    CMP     size, #(1 << NSET) - 1        ;是否过大?
    BHI     %FT1                          ;退出并返回丢弃者基

    IF "$op" = "Icache"
        MRC     p15, 0, c9f, c9, c0, 1    ;获取 I-cache 的锁定位
    ENDIF
    IF "$op" = "Dcache"
        MRC     p15, 0, c9f, c9, c0, 0    ;获取 D-cache 的锁定位
    ENDIF

    AND     tmp, c9f, #0xf                ;tmp = L 位的状态
    MOV     tmp1, #1
    TST     c9f, tmp1                     ;测试锁定位 0
    MOVNE   tmp1, tmp1, LSL #1
    TSTNE   c9f, tmp1                     ;测试锁定位 1
    MOVNE   tmp1, tmp1, LSL #1
    TSTNE   c9f, tmp1                     ;测试锁定位 2
    MOVNE   tmp1, tmp1, LSL #1
    BNE     %FT1                          ;错误:没有可取的路
    CMP     size, #0                      ;无锁定请求
    BEQ     %FT1                          ;退出返回 size = 0

    MVN     tmp1, tmp1                    ;选择 L 位
    AND     tmp1, tmp1, #0xf              ;清除寄存器中 L 位以外的其它位

```

```

BIC          c9f, c9f, #0xf          ;构造 c9f
ADD          c9f, c9f, tmp1

IF "$ op" = "Icache"
    MCR      p15, 0, c9f, c9, c0, 1    ;设置锁定 I 页
ENDIF
IF "$ op" = "Dcache"
    MCR      p15, 0, c9f, c9, c0, 0    ;设置锁定 D 页
ENDIF

5
IF "$ op" = "Icache"
    MCR      p15, 0, adr, c7, c13, 1    ;装载指令 cache 行
    ADD      adr, adr, #1 << CLINE      ;cache 行地址加 1
ENDIF
IF "$ op" = "Dcache"
    LDR      tmp1, [adr], #1 << CLINE    ;装载数据 cache 行
ENDIF

SUBS         size, size, #1            ;cline = - 1
BNE          %B75                      ;在各 cache 行进行循环

MVN          tmp1, c9f                ;锁定选中的 L 位
AND          tmp1, tmp1, ~#0xf         ;将屏蔽非 L 位
ORR          tmp, tmp, tmp1            ;与原 L 位合并
BIC          c9f, c9f, #0xf           ;清零所有的 L 位
ADD          c9f, c9f, tmp             ;设置 c9f 的 L 位

IF "$ op" = "Icache"
    MCR      p15, 0, adr, c9, c0, 1    ;设置 I-cache 锁定位
ENDIF
IF "$ op" = "Dcache"
    MCR      p15, 0, adr, c9, c0, 0    ;设置 D-cache 锁定位
ENDIF

1
MOV          r0, tmp1                  ;返回分配的路
MOV          pc, lr
MEND

```

```

lockDCache
    CACHELOCKBYLBIT Dcache
lockICache
    CACHELOCKBYLBIT Icache
ENDIF

```

接下来的 7 行检查 c9f 寄存器,以决定是否有一个路用来存储数据和代码。如果没有,则退出该例程;如果有,则在接下来的 4 行中改变 c9f 寄存器的值,以选择用于锁定数据的路。接着 c9f 寄存器被用在 MCR 指令中来选择路。

这样,程序进入把锁定的代码和数据填满 cache 的循环。如果要在 I-cache 中锁定代码,则将执行预取 I-cache 行命令;如果锁定外部存储器中的数据,则程序将清理、清除并向 D-cache 中装载新的 cache 行。

宏在退出返回时将所保存的原来的 cache 锁定位与新锁定的页融合,根据结果创建新的 c9f 寄存器。宏在 MCR 指令中使用 c9f 寄存器来设置 CP15;c9;c0 cache,锁定寄存器的 L 位。

最后,2 次使用宏 CACHELOCKBYBIT 创建 lockDCache 和 lockICache 函数。

12.6.4 在 Intel XScale SA-110 中锁定 cache 行

Intel XScale 处理器也有在 cache 中锁定代码和数据的能力。这要求使用一组 CP15;c9 cache 锁定命令,见表 12.13。CP15;c9;c2 寄存器的格式见图 12.15。另外还须使用在例 12.5 中清理 D-cache 时用的 CP15;c7,分配 D-cache 行命令,该命令见表 12.7。

表 12.13 在 Intel XScale 处理器 cache 中锁定代码和数据的取和分配命令

命 令	MRC 和 MCR 指令
取并锁定 I-cache 行	MCR p15,0,Rd,c9,c1,0
解锁指令 cache	MCR p15,0,Rd,c9,c1,1
读数据 cache 锁定寄存器	MRC p15,0,Rd,c9,c2,0
写数据 cache 锁定寄存器并置位/清零锁定模式	MCR p15,0,Rd,c9,c2,0
解锁 D-cache	MCR p15,0,Rd,c9,c2,1

在 Intel XScale 处理器中,cache 中的每个组都有一个轮转指针。每当有新的 cache 行在 cache 中被锁定时,该指针顺序增加。在一个组的 32 个 cache 行中,可以有最多 28 个 cache 行被锁定。若在一个组中试图锁定超过 28 个 cache 行,则会使该 cache 行被分配,但不能在 cache 中被锁定。

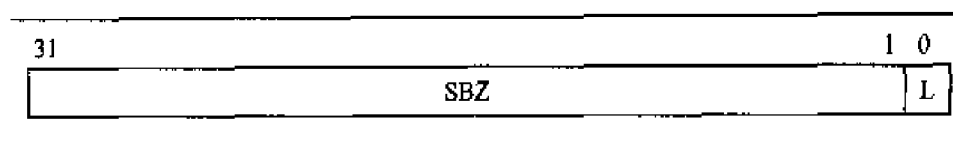


图 12.15 D-cache 锁定寄存器 CP15:c9:c2 格式

Intel XScale 处理器有 2 种方法在 D-cache 中锁定数据:第 1 种方法只是简单地将主存位置锁定在 D-cache 中;第 2 种方法使用分配 cache 行命令,将 cache 中的一部分配置成数据 RAM,这样,这部分分配的 cache 没有被初始化,需要处理器内核对它执行写操作来包含有效数据。在这里的例子中将存储器初始化为零。

【例 12.9】 例程的第一部分定义了宏 CACHELOCKREGION 中使用的寄存器。宏中也使用了头文件 cache.h 中定义的常量。

宏首先调整地址 adr 到 cache 行地址,并决定承载代码所需要的 cache 行数。

如果要在 D-cache 中锁定数据,那么例子中接下来的几行代码排空写缓冲器,并把 D-cache 解锁。在 D-cache 中锁定数据,必须在锁定一个 D-cache 行之前使用解锁命令。宏通过向 CP15:c9:c2:0 寄存器中写入 1 来将该位置位。

这时,程序进入循环,向 cache 中填充锁定的数据和代码。如果程序要在 I-cache 中锁定代码,那么将会执行锁定 I-cache 行命令;如果要锁定来自外部存储器的数据,那么程序将清理 cache,清除 cache 并向 D-cache 中装载新的 cache 行;如果要创建数据 RAM,那么程序会分配一个 D-cache 行,并且排空写缓冲器,以防止试图锁定超过 28 组数据而引起的错误。接下来使用 STRD 指令将 cache 行初始化为 0。

如果是锁定 D-cache 中的数据,那么宏在返回退出时,把 cache 装载 CP15 寄存器上的锁定位清零。

```

IF {CPU} = "XSCALE"
    EXPORT lockICache
    EXPORT lockDCache
    EXPORT lockDCacheRAM
    INCLUDE cache.h

```

adr	RN 0	;代码和数据的当前地址
size	RN 1	;存储器字节数
tmp	RN 2	;CPU15:c9:c2 寄存器/STRD reg0
tmp1	RN 3	;LDR 使用的临时寄存器/STRD reg1
MACRO		
CACHELOCKREGION \$ op		

```

ADD      size, adr, size      ;size = 末地址
BIC      adr, adr, #(1 << CLINE) - 1 ;与 CLINE 对齐
MOV      tmp, #(1 << CLINE) - 1 ;临时 CLINE 屏蔽
TST      size, tmp           ;CLINE 末尾有碎片?
SUB      size, size, adr      ;补上对齐字节数
MOV      size, size, lsr #CLINE ;转换大小 2 # CLINE
ADDNE    size, size, #1       ;增加 CLINE, 以承载碎片

```

```

CMP      size, #0            ;无锁定请求
BEQ      %FT1                ;退出返回 size = 0

```

```

IF      "$ op" = "Dcache" ;LOR, "$ op" = "DcacheRAM"
    MCR      p15, 0, adr, c7, c10, 4      ;排空写缓冲
    MOV      tmp, #1
    MCR      p15, 0, tmp, c9, c2, 0        ;解锁数据 cache
    CPWAIT
    MOV      tmp, #0                      ;偶数字清零
ENDIF
IF      "$ op" = "DcacheRAM"
    MOV      tmp1, #0                    ;初始化奇数字为 0
ENDIF

```

5

```

IF      "$ op" = "Icache"
    MCR      p15, 0, adr, c9, c1, 0        ;锁定 I-cache 行
    ADD      adr, adr, #1 << CLINE
ENDIF
IF      "$ op" = "Dcache"
    MCR      p15, 0, adr, c7, c10, 1      ;清理带脏位的 cache 行
    MCR      p15, 0, adr, c7, c6, 1       ;清除 D-cache 行
    LDR      tmp1, [adr], #1 << CLINE     ;装载数据 cache 行
ENDIF
IF      "$ op" = "DcacheRAM"
    MCR      p15, 0, adr, c7, c2, 5        ;分配 D-cache 行
    MCR      p15, 0, adr, c7, c10, 4      ;排空写缓冲
    STRD     tmp, [adr], #8                ;init 2 zero & adr = + 2

```

```

        STRD    tmp, [adr], #8           ;init 2 zero & adr = +2
        STRD    tmp, [adr], #8           ;init 2 zero & adr = +2
        STRD    tmp, [adr], #8           ;init 2 zero & adr = +2
    ENDIF
    SUBS        size, size, #1
    BNE         %BT5

    IF "$ op" = "Dcache" ;LOR; "$ op" = "DcacheRAM"
        MCR      p15, 0, adr, c7, c10, 4      ;排空写缓冲
        MOV      tmp, #0
        MCR      p15, 0, tmp, c9, c2, 0        ;锁定 data cache,tmp = 0 here
        CPWAIT
    ENDIF
1
    MOV         r0, #0
    MOV         pc, lr
    MEND

lockICache
        CACHELOCKREGION lcache
lockDCache
        CACHELOCKREGION Dcache
lockDCacheRAM
        CACHELOCKREGION DcacheRAM
    ENDIF

```

最后,3次使用宏创建 lockICache,lockDCache 和 lockDCacheRAM 函数。

12.7 cache 与软件性能

遵循一些简单的规则,将有助于利用 cache 的结构优势来编写代码。

存储器系统中的许多区域,通常都把 cache 和写缓冲器两者都使能,从而最大限度地利用 cache 体系结构的优点来缩短平均访存时间。有关存储系统的不同部分、cache 配置和写缓冲器操作的更多信息,可参见后续章节。如果使用带有存储器保护单元(MPU)的 ARM 处理器内核,则详细内容可参考第 13 章;如果使用带有存储器管理单元(MMU)的 ARM 处理器内核,则可参考第 14 章。

如果把存储器映射的外设配置成使用 cache 或写缓冲器,那么通常会产生问题。所以

最好将它们配置成不使用 cache,并且不使用写缓冲器,这就强制处理器在每次访问时都去读外设端口,而不是从 cache 中读取陈旧的信息。

应尽可能将经常访问的数据顺序存放在主存中,因为从主存中获取一个新的数据的代价等同于填充整个 cache 行。如果一个 cache 行中的数据在被替换出 cache 之前只使用过一次,那么系统的性能就比较低。应尽可能地把数据放在同一个 cache 行中,以提高 cache 命中率,因为这样可以充分利用局部性原理,从而形成更多的 cache 命中。最重要的一点是,要在主存中把一个共用例程所访问的数据尽量紧靠在一起。

应尽可能组织数据,使读、处理和写都在 cache 行尺寸的块中完成,并使主存块地址的低位与 cache 行的起始地址匹配。

最通用的做法是使代码尽量小,并将相关的数据分组放在一起。代码尺寸越小,cache 效率越高。

在 cache 系统中,使用链表会降低程序性能,因为查表会导致很高的 cache 失效率。与从顺序数组中访问数据相比,从链表访问数据,程序将以更加随机的形式取数据。这一点在查找任何无序表时都须考虑。选用何种数据查找方法,可能需要对系统性能进行分析。

然而,除了编写高效使用 cache 的代码之外,不要忘记还有许多其它的因素在改善系统性能上有更大的作用,高效的编程技巧详见第 5 章和第 6 章。

12.8 总 结

441

cache 是一个放置在处理器和主存之间的小容量高速存储器阵列。它是一个存储部分最近访问的主存内容的缓冲。相比于系统存储器,处理器在可能的情况下更优先使用 cache 存储器,以改善系统的平均性能。

写缓冲器是一个位于处理器内核与主存之间的非常小的先进先出(FIFO)存储器,它可以把处理器内核与 cache 存储器从低速的主存写操作中解脱出来。

局部性原理说明,程序在执行过程中会频繁运行小范围的循环代码,而这些代码会对数据存储器中的局部数据反复访问。它解释了为什么使用带 cache 的内核后,系统的平均性能会显著改善。

ARM 组织(ARM community)使用了许多条目来描述 cache 体系结构的特性。为了方便,这里给出了表 12.14,它列出了当前所有带 cache 的 ARM 内核的特性。

表 12.14 带 cache ARM 内核的特性

内 核	cache 类型	cache 大小/KB	cache 行 大小/word	相 联	位 置	是否支持 cache 锁定	写缓冲器 大小/word
ARM720T	统 一	8	4	4-way	逻 辑	否	8
ARM740T	统 一	4 或 8	4	4-way	逻 辑	是 1/4	8
ARM920T	分 离	16/16D+I	8	64-way	逻 辑	是 1/64	16
ARM922T	分 离	8/8D+I	8	64-way	逻 辑	是 1/64	16
ARM940T	分 离	4/4D+I	4	64-way	逻 辑	是 1/64	8
ARM926EJ-S	分 离	4-128/4-128D+I	8	4-way	逻 辑	是 1/4	16
ARM946E-S	分 离	4-128/4-128D+I	4	4-way	逻 辑	是 1/4	4
ARM1022E	分 离	16/16D+I	8	64-way	逻 辑	是 1/64	16
ARM1026EJ-S	分 离	4-128/4-128D+I	8	4-way	逻 辑	是 1/4	8
Intel StrongARM	分 离	16/16D+I	4	32-way	逻 辑	否	32
Intel XScale	分 离	32/32D+I	8	32-way	逻 辑	是 1/32	32
		2D	8	2-way	逻 辑	否	

cache 行是 cache 的基本组成单位,包含 3 部分:目录存储段(directory store)、数据项段(data section)和状态信息段(status information)。cache 标签(cache-tag)是一个目录记录项,指示一个 cache 行是从主存中的什么地方被装载的。在 cache 中通常有 2 个状态位:有效位和脏位。当相关的 cache 行包含有效的存储器内容时,有效位被置位;当 cache 使用回写策略并且有新的数据写入到 cache 行时,脏位被置位。

cache 的位置可以在 MMU 之前或之后,有物理 cache 和逻辑 cache 之分。逻辑 cache 被放置在处理器内核与 MMU 之间,在虚拟地址空间访问代码和数据。物理 cache 被放置在 MMU 和主存之间,使用物理地址访问代码和数据。

直接映射 cache 是一种非常简单的 cache 结构,每一个主存地址都对应惟一的 cache 地址。直接映射 cache 经常会导致“颠簸”。为了减少“颠簸”,将 cache 分成容量相等的较小单元,这种小单元被称作路(way)。使用路为一个主存单元在 cache 中提供了对应的多个存储位置。这种 cache 被称作组相联 cache。

内核总线的体系结构决定了 cache 系统的设计。冯·诺依曼结构使用统一 cache 存储代码和数据。哈佛结构使用分离 cache:一个 cache 用于指令,另一个 cache 用于数据。

cache 替换策略决定了在访问 cache 失效时,哪一个 cache 行将被替换出 cache。配置策略决定 cache 控制器使用什么算法在当前 cache 存储器的组中选择一个 cache 行。被选中作替换的 cache 行被称作丢弃者。带 cache 的 ARM 内核使用两种替换策略:伪随机法和

轮转法。

向 cache 中写数据有两种策略:如果控制器只更新 cache 存储器,称为回写(*writeback*)策略;如果 cache 控制器既写 cache,又写主存,则称为直写(*writethrough*)策略。

当 cache 失效时,cache 控制器使用两种策略分配一个新的 cache 行:读分配策略在数据由主存中读出时分配 cache 行;写分配策略在向主存中写数据时分配 cache 行。

ARM 使用术语清理(*clean*)表示将 D-cache 中的数据写回到主存中。ARM 使用术语清除(*flush*)表示使 cache 中的内容无效(作废)。

有些 ARM 内核提供 cache 锁定功能。锁定允许代码和数据被装载到 cache,并被标记为非替换的。

本章还提供了一些示例代码,显示了如何清理、清除 cache,以及在 cache 中锁定代码和数据。

第 13 章

存储器保护单元 MPU

- 受保护的区域
- 初始化 MPU, cache 和写缓冲器
- MPU 系统示例
- 总 结

一些嵌入式系统使用多任务的操作或控制系统。在这种系统里,必须保证正在运行的任务不破坏其它任务的操作。防止系统资源和其它任务受非法访问的工作称为保护,这也是本章将要讨论的主要内容。

系统资源的访问控制有两种方法:无硬件保护(简称无保护)和有硬件保护(简称受保护)。无保护的系统仅靠软件来保护系统资源;受保护的系统靠硬件和软件两者来保护系统资源。在具体的控制系统中使用哪一种方法,取决于处理器的性能和该控制系统的需求。

在无保护的嵌入式系统中,没有专门用于存储器和外围设备访问控制的硬件。在这种系统中,任何一个任务都可能破坏其它任务的状态,因此每个任务在访问系统资源时都必须与其它所有的任务进行协调。当一个任务忽略其它任务环境的访问限制时,这种协调机制可能会导致任务失败。

这里是无保护系统中可能出现的一个任务失败的例子:当读/写一个通信用的串口寄存器时,如果一个任务正在使用串口,则它没有办法来防止其它任务使用同一个串口。因此,若要成功使用该串口,则必须通过一个访问该串口的系统调用来协调。但使用这些调用任务的非授权访问,很容易破坏经过该串口的通信。因此资源的不合理使用也许是不可避免的,或者说它本质上就存在。

相反,受保护系统有专门的硬件来检测和限制系统资源的访问。它能保证资源的所有权,任务需要遵守一组由操作环境定义的、由硬件维护的规则,在硬件级上授予监视和控制资源程序的特殊权限。受保护系统主动防止一个任务使用其它任务的资源。因此使用硬件主动监视系统比协调加强的软件例程,提供了更好的保护。

ARM 的很多处理器配备了有效保护系统资源的硬件,或者通过存储器保护单元 MPU (Memory Protection Unit),或者通过存储器管理单元 MMU (Memory Management Unit)。带有 MPU 的处理器核是本章讨论的重点,它对一些由软件定义的区域提供硬件保护。带有 MMU 的处理器核是下一章讨论的重点,它提供硬件保护并增加了虚拟存储器功能。

在受保护的系统中,主要有两类资源需要监视:存储器系统和外围设备。ARM 的外围设备通常都映射到存储器中,因此 MPU 就使用同样的方法来保护这两类资源。

ARM 的 MPU 使用区域(region)来管理系统保护。区域是与一个存储空间相关联的一组属性,处理器核将这些属性保存在协处理器 CP15 的一些寄存器里,并用 0~7 的号码标识每个区域*。

区域的存储器边界通过两个属性进行配置:起始地址和大小。大小可以是 4 KB~4 GB 的任何 2 的乘幂。另外,操作系统可以为这些区域分配更多的属性:访问权限、cache 和写缓冲器策略。存储器中对区域的访问可以是读/写、只读或不可访问,并基于当时的处理器模

* 在带 MPU 的系统中,区域反映了一个存储空间的属性,同时也代表了一个具有特定属性的逻辑存储空间。以下区域一词的大部分含义皆为后者。——译者注

式——管理模式或用户模式,还有一些附加的权限。区域还有控制 cache 和写缓冲器属性的 cache 写策略。例如,可以设置一个区域使用直写(或称写直达法, writethrough)策略访问存储器,而另一个区域则以无 cache 和无写缓冲方式操作。

当处理器访问主存的一个区域时,MPU 比较该区域的访问权限属性和当时的处理器模式。如果请求符合区域访问标准,则 MPU 允许内核读/写主存;如果存储器请求导致存储器访问违例,则 MPU 产生一个异常信号。

异常信号被传送到处理器核。处理器核执行一个异常向量,然后跳转到异常处理程序,以响应该异常信号。异常处理程序可以判断异常类型为预取指或数据中止,然后根据异常类型,跳转到相应的服务例程。

要实现一个受保护系统,控制系统对主存中的不同块定义若干区域。一个区域可以被创建一次,然后一直作用到嵌入式系统结束;也可以被临时创建来满足一个特殊操作的需要,随后就被删除。下一节的主题就是如何来分配和创建区域。

13.1 受保护的区域

目前有 4 种 ARM 核包含 MPU:ARM740T,ARM940T,ARM946E-S 和 ARM1026EJ-S。ARM740T,ARM946E-S 和 ARM1026EJ-S 包含 8 个受保护区域;ARM940T 包含 16 个受保护区域(参看表 13.1)。

ARM740T,ARM946E-S 和 ARM1026EJ-S 使用统一的指令和数据区域——指令区域和数据区域使用相同的寄存器进行定义,这些寄存器设置区域的起始地址和大小。在 ARM946E-S 和 ARM1026EJ-S 中,为指令和数据访问分别配置存储器访问权限和 cache 策略;在 ARM740T 中,为指令和数据存储器分配相同的访问权限和 cache 策略。区域与内核是冯·诺依曼(von Neumann)结构还是哈佛(Harvard)结构无关。每个区域通过 0~7 的号码来标识和引用。

因为 ARM940T 采用独立的区域来控制指令和数据存储器,所以可以为指令和数据区域定义不同的区域起始地址和大小。指令和数据区域的分离导致在这种核中增加了 8 个区域。虽然在 ARM940T 中标识区域的号码仍然是 0~7,但是每个号码对应有两个区域:一个数据区域和一个指令区域。

有关区域的一些属性如下:

- 区域可以相互重叠;
- 每个区域都分配有一个优先级,该优先级与分配给区域的权限无关;
- 当区域重叠时,具有最高优先级的区域的属性可以覆盖其它区域的属性,优先级仅作用于重叠部分的地址;

- 区域的起始地址必须是其大小的倍数；
- 区域的大小可以是 4 KB~4 GB 的任何 2 的乘幂,即为下面的任何值:4 KB,8 KB,16 KB,32 KB,64 KB,...,2 GB,4 GB;
- 访问所定义区域外的存储器将产生异常。如果是内核预取指令,则 MPU 产生预取指中止异常;如果是存储器数据请求,则产生数据中止异常。

表 13.1 带保护单元的 ARM 核概要

ARM 核	区域数目	指令和数据区域是否分离	指令和数据区域配置是否分离
ARM740T	8	否	否
ARM940T	16	是	是
ARM946E-S	8	否	是
ARM1026EJ-S	8	否	是

13.1.1 重叠区域

当存储空间的某部分被分配给一个以上的区域时,会发生区域重叠。重叠区域比非重叠区域在分配访问权限时有更大的灵活性。

作为区域重叠的一个例子,假设有一个小的嵌入式系统,它有 256 KB 的可用存储空间,起始地址为 0x00000000,现在须保护一块特权系统空间,用户(user)模式下的程序不能对其进行读/写操作。这块特权空间的代码、数据和堆栈在一个 32 KB 的区域内,这个区域的最前面是向量表,起始地址为 0x00000000,剩下的存储空间作为用户空间。

有了重叠区域,系统使用 2 个区域:一个 256 KB 的用户区域和一个 32 KB 的特权区域(见图 13.1)。给特权区域 1 分配较高的优先级,因为它的属性必须优先于用户区域 0。

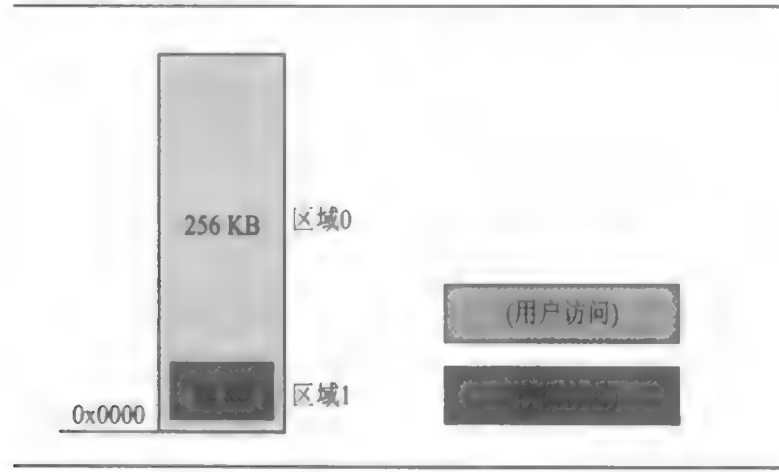


图 13.1 建立重叠区域

13.1.2 背景区域

重叠区域的另一个有用特征是作为背景区域。用来给一块大存储空间分配相同属性的低优先级区域。其它具有较高优先级的区域与该背景区域的某部分重叠,用来改变已定义的背景区域的较小子集的属性。这样,具有较高优先级的区域可以改变背景区域属性的子集。背景区域可以用来保护一些睡眠状态的存储空间,使其不受非法访问,而此时由另一个不同区域控制下的背景区域的其它部分可以处于活跃状态。

例如,一个嵌入式系统定义了一个大的特权背景区域,可以让一个较小的非特权的区域与这个背景区域的某部分重叠。这个较小区域的位置可以在该背景区域的不同位置,以代表不同的用户空间。当系统将这个较小的用户区域从一个位置移动到另一个位置时,早先被覆盖的空间由背景区域所保护。因此用户区域就像窗口一样,允许访问特权背景区域的不同部分,但它只有用户级属性(见图 13.2)。

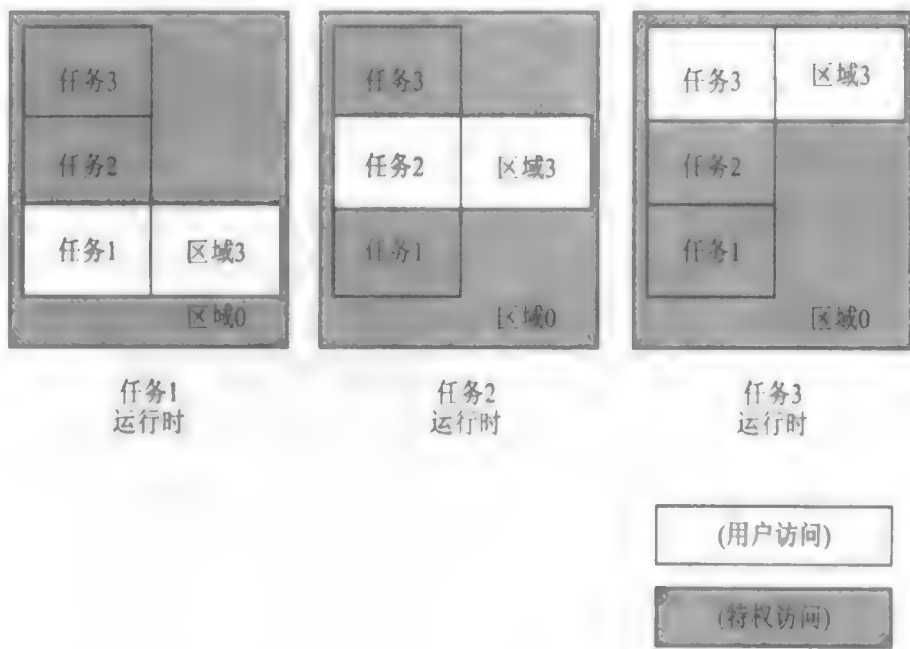


图 13.2 用背景区域控制任务访问

图 13.2 表示一个简单的 3 个任务的保护模式。区域 3 定义活跃任务的保护属性,背景区域 0 控制睡眠态任务空间的访问。当任务 1 运行时,背景区域保护任务 2 和任务 3 的空间不受任务 1 访问;当任务 2 运行时,任务 1 和任务 3 的空间被保护;最后,当任务 3 运行时,任务 1 和任务 2 的空间被保护。这样工作的原因是区域 3 比区域 0 有较高的优先级,尽管区域 0 有较高的访问权限。

在本章最后的例程代码中用一个背景区域来演示一个简单的多任务保护模式。

13.2 初始化 MPU, cache 和写缓冲器

为了初始化 MPU, cache 和写缓冲器, 控制系统必须定义在操作目标平台时所需的保护区域。

在启用存储器保护单元之前, 必须至少定义一个数据区域和一个指令区域, 而且必须在启用 cache 和写缓冲器之前(或同时)启用存储器保护单元。

控制系统通过设置 CP15 的主(primary)寄存器 c1, c2, c3, c5 和 c6 来配置 MPU。表 13.2 列出了用来控制 MPU 操作的主寄存器, 寄存器 c1 是主要的控制寄存器。

表 13.2 用来控制 MPU 的协处理器寄存器

功 能	主寄存器	次寄存器
系统控制	c1	c0
区域的 cache 属性	c2	c0
区域的写缓冲器属性	c3	c0
区域的访问权限	c5	c0
区域的大小和位置	c6	c0~c7

通过配置寄存器 c2 和 c3 来设置区域的 cache 和写缓冲器的属性, 寄存器 c5 控制区域的访问权限, 在寄存器 c6 里有 8 或 16 个次(secondary)寄存器, 用来定义每个区域的大小和位置。在 ARM740T, ARM940T, ARM946E-S 和 ARM1026EJ-S 中还有其它配置寄存器, 但是其用法不包括 MPU 的基本操作。有关协处理器 CP15 的寄存器的用法, 请参见 3.5.2 小节。

初始化 MPU, cache 和写缓冲器需要以下步骤:

- ① 使用 CP15:c6 来定义指令和数据区域的大小和位置;
- ② 使用 CP15:c5 来设置每个区域的访问权限;
- ③ 分别使用 CP15:c2 和 CP15:c3 来设置每个区域的 cache 和写缓冲器属性;
- ④ 使用 CP15:c1 来使能 cache 和 MPU。

在描述完用来配置每个寄存器的协处理器 CP15 命令以后, 都有一小节描述每个步骤, 也有例程代码来说明在初始化过程中, 完成该步骤的程序中所用到的命令。

13.2.1 定义区域的大小和位置

嵌入式系统通过写 8 个次寄存器 CP15:c6:c0:0~CP15:c6:c7:0 来定义每个区域的大小和地址范围。每个协处理器次寄存器号映射到对应的区域标识号。

每个区域的起始地址必须对齐到其大小的整数倍。比如,一个区域的大小是 128 KB,其起始地址可以是 0x20000 的整数倍的任何数。区域的大小可以是 4 KB~4 GB 的 2 的任意乘幂。

图 13.3 和表 13.3 说明 CP15:c6:c0~CP15:c6:c7 的 8 个次寄存器的位域和格式。最高位域[31:12]保存起始地址,起始地址必须是大小域[5:1]的整数倍。E 位域[0]使能或禁用该区域。也就是说,区域可以被定义和禁用,因此其属性直到使能位被置位才生效。CP15:c6 次寄存器中未定义的位必须置为 0。

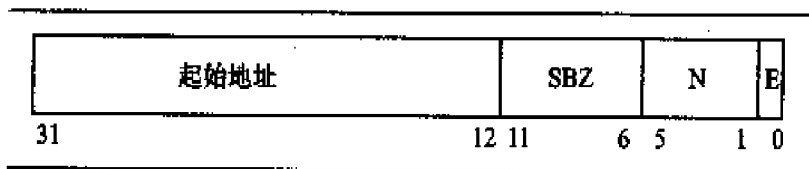


图 13.3 设置区域的大小和位置的 CP15:c6 寄存器格式

表 13.3 CP15:c6:c0~CP15:c6:c7 的寄存器位域描述

域 名	位 域	注 释
起始地址	[31:12]	比 4 KB 大的地址必须是[5:1]中表示的大小的倍数
SBZ	[11:6]	必须置为 0
N	[5:1]	区域的大小尺寸为 2^{N+1} , 这里 $11 \leq N \leq 31$
E	[0]	区域使能, 1=使能, 0=禁用

可以用公式 $size=2^{N+1}$ 或查找表 13.4 的值来定义区域的大小,在 CP15:c6 寄存器的大小位域中填入指数 N 来设置大小。硬件设置时,限制 N 的值为 11~31 的任意值,表示大小为 4 KB~4 GB。确定了区域的大小后,区域的起始地址可以用公式算出(大小的整数倍),也可以查表 13.4。区域的大小和起始地址与系统的存储器映射和控制系统必须保护的空间有关。本章最后的示例系统说明在给定的系统存储器映射中如何建立区域。

表 13.4 区域大小编码

大 小	十进制值	二进制值	大 小	十进制值	二进制值
4 KB	11	01011	8 MB	22	10110
8 KB	12	01100	16 MB	23	10111
16 KB	13	01101	32 MB	24	11000
32 KB	14	01110	64 MB	25	11001
64 KB	15	01111	128 MB	26	11010
128 KB	16	10000	256 MB	27	11011
256 KB	17	10001	512 MB	28	11100
512 KB	18	10010	1 GB	29	11101
1 MB	19	10011	2 GB	30	11110
2 MB	20	10100	4 GB	31	11111
4 MB	21	10101			

ARM740T, ARM946E-S 和 ARM1026EJ-S 处理器有 8 个区域, 通过写 CP15: c6: cX 的次寄存器来设置区域的大小和位置。比如, 将区域 3 的位置和大小设置成起始地址为 0x300000、大小为 256 KB 的指令序列如下:

```
MOV r1, #0x300000      ;设置起始地址
ORR r1, r1, #0x11 << 1 ;设置大小为 256 KB
MCR p15, 0, r1, c6, c3, 0
```

先在内核的寄存器 r1 中填入所需的位域值, 然后用 MCR 指令将 r1 的值写入 CP15 的次寄存器。

ARM940T 有 8 个指令区域和 8 个数据区域, 需要一个附加的操作码 2 修正值来选择是指令区域, 还是数据区域。若为数据区域, 则操作码 2 为 0; 若为指令区域, 则操作码 2 为 1。

例如, 需要 2 条 MRC 指令才能读取数据区域和指令区域的大小和位置, 一条读数据区域的大小和位置, 一条读指令区域的大小和位置。读取区域 5 的大小和起始地址的指令如下:

```
MRC p15, 0, r2, c6, c5, 0 ;r2 = 数据区域 5 的起始地址和大小
MRC p15, 0, r3, c6, c5, 1 ;r3 = 指令区域 5 的起始地址和大小
```

第一条指令将数据区域 5 的大小和起始地址装载到内核的寄存器 r2 中, 第二条指令将指令区域 5 的大小和起始地址装载到内核的寄存器 r3 中。ARM940T 是目前惟一的指令区域和数据区域分离的处理器核。

ARM 嵌入式系统开发

【例 13.1】说明怎样设置区域的起始地址、大小和使能位。

例程 regionSet 的 C 函数原型如下：

```
void regionSet(unsigned region, unsigned address, unsigned sizeN, unsigned enable);
```

这个例程有 4 个无符号整型的输入参数：要配置的区域、区域的起始地址、编码以后的区域的大小 sizeN 以及区域是使能还是禁止。在改变区域的属性时，最好是先禁用它，然后在改变完成以后再重新使能它。

为了使这个例程可以在全部 4 个带 MPU 的处理器上运行，可以通过配置指令区域和数据区域的大小和起始地址信息来统一 ARM940T 的区域空间。为了实现这个功能，编写的宏 SET_REGION 包含两部分：一部分专为 ARM940T；另一部分为其它核。这样，相同的例程就可以支持 4 个带 MPU 的核了。

```
# if defined( __TARGET_CPU_ARM940T)
# define SET_REGION(REGION_NUMBER) \
    /* 设置数据区域的起始地址和大小 */ \
    __asm{MCR p15, 0, c6f, c6, c ## REGION_NUMBER, 0} \
    /* 设置指令区域的起始地址和大小 */ \
    __asm{MCR p15, 0, c6f, c6, c ## REGION_NUMBER, 1}
# endif

# if defined(__TARGET_CPU_ARM946E_S) | \
    defined(__TARGET_CPU_ARM1026EJ_S)
# define SET_REGION(REGION_NUMBER) \
    /* 设置区域的起始地址和大小 */ \
    __asm{MCR p15, 0, c6f, c6, c ## REGION_NUMBER, 0}
# endif

void regionSet(unsigned region, unsigned address,
               unsigned sizeN, unsigned enable)
{
    unsigned c6f;

    c6f = enable | (sizeN << 1) | address;
    switch (region)
    {
        case 0: { SET_REGION(0);break;}
        case 1: { SET_REGION(1);break;}
    }
```

```

case 2: { SET_REGION(2);break;}
case 3: { SET_REGION(3);break;}
case 4: { SET_REGION(4);break;}
case 5: { SET_REGION(5);break;}
case 6: { SET_REGION(6);break;}
case 7: { SET_REGION(7);break;}
default: { break;}
}
}

```

代码首先将起始地址、sizeN 和使能位等区域属性合并到一个无符号整型变量 c6f; 然后跳转到用宏 SET_REGION 创建的 8 个 regionSet 例程之一, SET_REGION 通过写 CP15:c6 次寄存器, 为定义的区域设置起始地址、大小和使能状态。

13.2.2 访问权限

有 2 组可用的访问权限机制: 标准组和扩展组。4 个核都支持标准组, 标准组有 4 级权限。ARM946E-S 和 ARM1026EJ-S 支持扩展组, 扩展组增加了额外的 2 级权限(见表 13.5)。扩展组 AP(access permission, 访问权限)位域编码支持 12 个增加的权限值, 现在只有 2 个值有定义。使用未定义的编码, 将导致不可预知的结果。

表 13.5 CP15 寄存器 5 访问权限

管理者	用户	标准 AP 值编码	扩展 AP 值编码
不可访问	不可访问	00	0000
读/写	不可访问	01	0001
读/写	只读	10	0010
读/写	读/写	11	0011
不可预知	不可预知	—	0100
只读	不可访问	—	0101
只读	只读	—	0110
不可预知	不可预知	—	0111
不可预知	不可预知	—	1000~1111

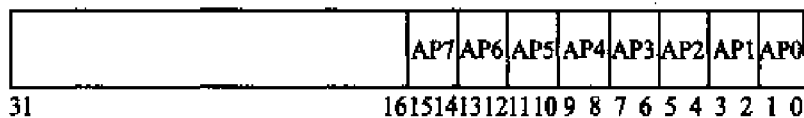
通过写 CP15:c5 的次寄存器来分配区域的访问权限。次寄存器 CP15:c5:c0:0 和 CP15:c5:c0:1 配置标准的 AP, 次寄存器 CP15:c5:c0:2 和 CP15:c5:c0:3 配置扩展的 AP。表 13.6 和图 13.4 说明 AP 寄存器的寄存器权限位分配。

表 13.6 标准和扩展访问权限寄存器 CP15:c5 的位域分配

区 域	标准 AP		扩展 AP	
	域 名	位 域	域 名	位 域
0	AP0	[1;0]	cAP0	[3;0]
1	AP1	[3;2]	eAP1	[7;4]
2	AP2	[5;4]	eAP2	[11;8]
3	AP3	[7;6]	eAP3	[15;12]
4	AP4	[9;8]	eAP4	[19;16]
5	AP5	[11;10]	eAP5	[23;20]
6	AP6	[13;12]	eAP6	[27;24]
7	AP7	[15;14]	eAP7	[31;28]

CP15:c5:c0标准指令区域AP

CP15:c5:c1标准数据区域AP



CP15:c5:c2扩展指令区域AP

CP15:c5:c3扩展数据区域AP

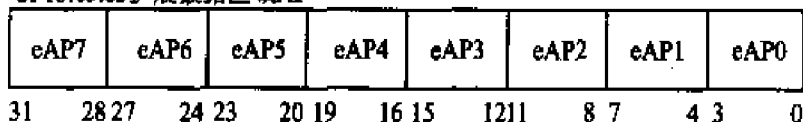


图 13.4 CP15 寄存器 5 访问权限寄存器格式

支持扩展权限的处理器也可以运行为标准权限编写的软件。有效的权限类型取决于对 CP15 AP 寄存器的最后一次写入：若最后一次写入的 AP 寄存器是标准 AP 寄存器，则内核使用标准权限；若最后一次写入的 AP 寄存器是扩展 AP 寄存器，则内核使用扩展权限。这样做的原因是，写标准 AP 寄存器也更新了扩展 AP 寄存器，即扩展 AP 区域的高位[2:3]被清除了。

当使用标准 AP 时，每个区域在寄存器 CP15:c5:c0:0 和 CP15:c5:c0:1 中有 2 位。CP15:c5:c0:0 设置数据 AP，CP15:c5:c0:1 设置指令区域。

读指令和数据空间的标准 AP，需要读 2 个寄存器。下面的 2 句 MRC 指令序列将数据区域的 AP 信息放在内核寄存器 r1 中，指令区域的 AP 信息放在寄存器 r2 中：

```
MRC p15, 0, r1, c5, c0, 0    ;数据区域标准 AP
MRC p15, 0, r2, c5, c0, 1    ;指令区域标准 AP
```

如果是扩展 AP,则每个区域在寄存器 CP15:c5:c0:2 和 CP15:c5:c0:3 中有 4 位。内核将 8 个区域的指令信息保存在一个寄存器中,数据信息保存在另一个寄存器中。CP15:c5:c0:2 设置数据区域的 AP,CP15:c5:c0:3 设置指令区域的 AP。

要获取指令和数据区域的扩展 AP,也需要读 2 个寄存器。下面的 2 句指令序列将数据区域的 AP 放在核寄存器 r3 中,指令区域的 AP 放在寄存器 r4 中:

```
MRC p15, 0, r3, c5, c0, 2    ;数据区域扩展 AP
MRC p15, 0, r4, c5, c0, 3    ;指令区域扩展 AP
```

下面将用 2 个例子来说明如何使用访问权限。一个说明标准 AP,另一个说明扩展 AP。这 2 个例子使用内嵌汇编代码来读/写 CP15 寄存器。

这里提供 2 个标准 AP 例程:regionSetISAP 和 regionSetDSAP,以设置区域的标准 AP 位。在 C 中调用时,使用下面的函数原型:

```
void regionSetISAP(unsigned region, unsigned ap);
void regionSetDSAP(unsigned region, unsigned ap);
```

第一个参数是区域号,第二个参数是 2 位的值,用来定义被区域控制的指令或数据空间的标准 AP。

【例 13.2】 2 个例程除了读/写不同的 CP15:c5 次寄存器外,几乎完全相同:一个是读/写指令寄存器,另一个是读/写数据寄存器。例程通过对 CP15:c5 寄存器进行简单的读—修改—写操作来设置指定区域的 AP,而不改变其它区域的配置。

```
void regionSetISAP(unsigned region, unsigned ap)
{
    unsigned c5f, shift;

    shift = 2 * region;
    __asm{ MRC p15, 0, c5f, c5, c0, 1 }           /* 装载标准数据 AP */
    c5f = c5f & ~(0x3 << shift);                 /* 清除原来的 AP 位 */
    c5f = c5f | (ap << shift);                    /* 设置新的 AP 位 */
    __asm{ MCR p15, 0, c5f, c5, c0, 1 }           /* 保存标准数据 AP */
}

void regionSetDSAP(unsigned region, unsigned ap)
{
```

```

unsigned c5f, shift;

shift = 2 * region;                                /* 设置位域宽度 */
__asm{ MRC p15, 0, c5f, c5, c0, 0 }                /* 装载标准指令 AP */
c5f = c5f & ~(0x3 << shift);                       /* 清除原来的 AP 位 */
c5f = c5f | (ap << shift);                         /* 设置新的 AP 位 */
__asm{ MCR p15, 0, c5f, c5, c0, 0 }                /* 保存标准指令 AP */
}

```

例程设置了特定区域的权限,先使用一个被移位的屏蔽码来清除 AP 位,然后用 ap 输入参数来设置 AP 位域。AP 位域的位置是区域号乘以权限位域的位数,即变量 shift。通过移位 ap 值和使用 OR 指令来修改 c5f 核寄存器,以设置位域值。

这里提供 2 个扩展 AP 例程:regionSetIEAP 和 regionSetDEAP,以设置区域的扩展 AP 位。它们的 C 函数原型如下:

```

void regionSetIEAP(unsigned region, unsigned ap);
void regionSetDEAP(unsigned region, unsigned ap);

```

第一个参数是区域号,第二个参数是 4 位的值,表示由区域控制的指令或数据存储空间的扩展 AP。

【例 13.3】 2 个例程除了读/写不同的 CP15:c5 次寄存器和 4 位宽的 AP 位域外,与标准 AP 例程几乎完全相同。

```

void regionSetIEAP(unsigned region, unsigned ap)
{
    unsigned c5f, shift;

    shift = 4 * region;                            /* 设置位域宽度 */
    __asm{ MRC p15, 0, c5f, c5, c0, 3 }            /* 装载扩展数据 AP */
    c5f = c5f & ~(0xf << shift);                   /* 清除原来的 AP 位 */
    c5f = c5f | (ap << shift);                     /* 设置新的 AP 位 */
    __asm{ MCR p15, 0, c5f, c5, c0, 3 }            /* 保存扩展数据 AP */
}

void regionSetDEAP(unsigned region, unsigned ap)
{
    unsigned c5f, shift;

    shift = 4 * region;                            /* 设置位域宽度 */

```

```
__asm{ MRC p15, 0, c5f, c5, c0, 2 }          /* 装载扩展指令 AP */
c5f = c5f & ~(0xf << shift);                /* 清除原来的 AP 位 */
c5f = c5f | (ap << shift);                  /* 设置新的 AP 位 */
__asm{ MCR p15, 0, c5f, c5, c0, 2 }          /* 保存扩展指令 AP */
}
```

每个例程都使用一个移位后的屏蔽码来清除 AP 位,然后用 ap 输入参数设置 AP 位域,以设置指定的区域权限。AP 位域的位置是区域号乘以权限位域的位数,即变量 shift 的值。通过移位 ap 值和使用 OR 指令来修改 c5f 核寄存器,以设置位域值。

13.2.3 设置区域的 cache 和写缓冲器属性

每个内核有 3 个 CP15 寄存器用来控制区域的 cache 和写缓冲器属性。其中 CP15:c2:c0:0 和 CP15:c2:c0:1 两个寄存器保存 D-cache 和 I-cache 区域属性;第 3 个寄存器,CP15:c3:c0:0 用于保存区域写缓冲器属性,并应用于存储器数据区域(详细内容参见图 13.5 和表 13.7)。

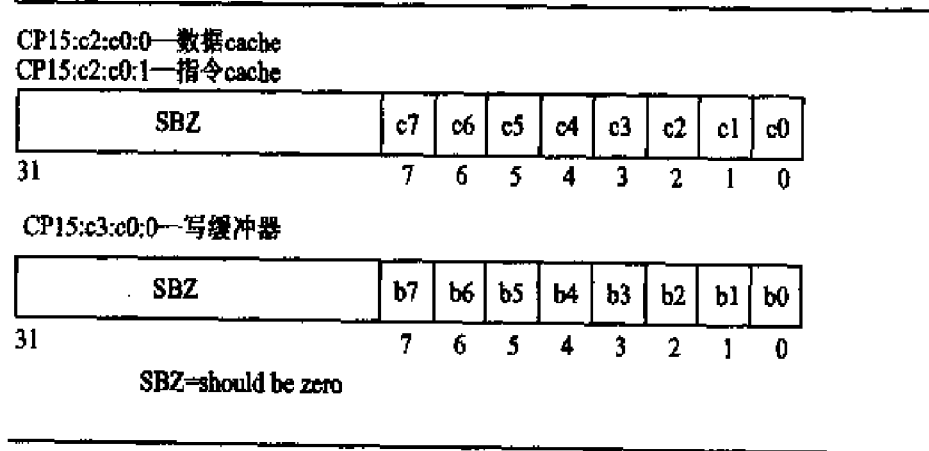


图 13.5 CP15:c2 cache 和 CP15:c3 写缓冲器区域寄存器

表 13.7 CP15:c2 和 CP15:c3 寄存器位域分配

区 域	区域 cache 域		区域写缓冲器域	
	域 名	位 域	域 名	位 域
0	c0	[0]	b0	[0]
1	c1	[1]	b1	[1]
2	c2	[2]	b2	[2]
3	c3	[3]	b3	[3]

续表 13.7

区 域	区域 cache 域		区域写缓冲器域	
	域 名	位 域	域 名	位 域
4	c4	[4]	b4	[4]
5	c5	[5]	b5	[5]
6	c6	[6]	b6	[6]
7	c7	[7]	b7	[7]

寄存器 CP15:c2:c0:1 包含所有 8 个指令区域的 cache 配置信息,寄存器 CP15:c2:c0:0 包含所有 8 个数据区域的 cache 配置信息。这 2 个寄存器使用相同的位域编码。

cache 位决定对区域内一个给定的地址是否使能 cache。在 ARM740T 和 ARM940T 中,cache 总是被查找,而不管 cache 位的状态。如果控制器找到一个有效的 cache 数据项,那么它就使用 cache 内的数据,而不使用外部存储器的数据。

基于这个 cache 特征,当 cache 策略从使用 cache 变为不使用 cache 时,必须清除并可能要清理区域的 cache。因此,MPU 控制系统每次改变 cache 写策略,从直写变为禁用 cache 时,都必须清除 cache;而从回写变为禁用 cache 时,都必须清理并清除 cache;而从回写变为直写时,必须清理 cache。清除和清理 cache 的例程见第 12 章。*

在 ARM946E—S 中,如果 cache 位被清除,则物理上存储在 cache 中的信息将不从 cache 中返回,而是直接访问外部存储器。这种设计减少了当 cache 被禁用时清除 cache 的次数,然而,原来区域的清理规则仍然适用。

寄存器 CP15:c3:c0:0 中的 8 个区域写缓冲器位,使能或禁用每个区域的写缓冲器(见图 13.5)。

当配置数据区域时,区域的 cache 位和写缓冲器位一起决定区域的策略。写缓冲器位有 2 个用途:使能或禁用区域的写缓冲器和设置区域的 cache 写策略。区域的 cache 位控制写缓冲器位的作用。当 cache 位为 0 时,写缓冲器位为 1,则使能写缓冲器;写缓冲器位为 0,则禁用写缓冲器。当 cache 位为 1 时,cache 和写缓冲器都被使能,此时写缓冲器位决定 cache 写策略。若写缓冲器位为 0,则区域使用直写策略;若写缓冲器位为 1,则区域使用回写策略。表 13.8 给出了 cache 和写缓冲器位的各种状态和其含义。直写和回写策略的详细内容见 12.3.1 小节。

* 关于清除、清理 cache 的含义请参考 12.6 节。——译者注

表 13.8 cache 和写缓冲器控制

指令 cache		数据 cache		
cache 位	区域属性	cache 位	写缓冲器位	区域属性
CP15:c2:c0;1		CP15:c2:c0;0	CP15:c3:c0;0	
0	不使用 cache	0	0	NCNB(不使用 cache,不使用写缓冲器)
1	使用 cache	0	1	NCB(不使用 cache,使用写缓冲器)
		1	0	WT(使用 cache,直写策略)
		1	1	WB(使用 cache,回写策略)

这里给出 2 个例程来示范使能和禁用 cache 和写缓冲器。这 2 个例程使用内嵌汇编来读/写 CP15 的寄存器。

通过合并 cache 和写缓冲器的控制为一个简单的例程调用,以简化系统配置。通过它们控制的写策略来引用数据 cache 位和写缓冲器位,指令 cache 位单独表示。从系统角度来看,为每个区域合并 cache 和写缓冲器为一个单独的值,容易将区域信息分组到一个区域控制块(将在 13.3.3 小节讨论)。

设置 cache 和写缓冲器的例程,称为 regionSetCB,在例 13.4 中表示,并有下面的 C 函数原型:

```
void regionSetCB(unsigned region, unsigned CB);
```

例程有 2 个输入参数:第 1 个参数 region,是区域号;第 2 个参数 CB,合并区域指令 cache 属性和数据 cache 与写缓冲器属性。第 2 个参数有格式,使用无符号整数的低 3 位:位[2]代表指令 cache 位,位[1]代表数据 cache 位,位[0]代表数据写缓冲器位。

【例 13.4】 设置 cache 和写缓冲器。

例程顺序设置数据写缓冲器位、数据 cache 位和指令 cache 位。对每个位,都是先读出 CP15 的寄存器;然后清除原来的位值,并设置新的位值;最后将值写回到 CP15 的寄存器。

```
void regionSetCB(unsigned region, unsigned CB)
{
    unsigned c3f, tempCB;

    tempCB = CB;
    __asm(MRC p15, 0, c3f, c3, c0, 0) /* 装载写缓冲器控制寄存器 */
    c3f = c3f & ~(0x1 << region); /* 清除原来的写缓冲器位 */
    c3f = c3f | ((tempCB & 0x1) << region); /* 设置新的写缓冲器位 */
    __asm(MCR p15, 0, c3f, c3, c0, 0) /* 保存写缓冲器控制信息 */
}
```

```

tempCB = CB >> 0x1;                                /* 移位至数据 cache 位 */
__asm{MRC p15, 0, c3f, c2, c0, 0}                  /* 装载数据 cache 控制寄存器 */
c3f = c3f & ~(0x1 << region);                      /* 清除原来的数据 cache 位 */
c3f = c3f | ((tempCB & 0x1) << region);             /* 设置新的数据 cache 位 */
__asm{MCR p15, 0, c3f, c2, c0, 0}                  /* 保存数据 cache 控制信息 */

tempCB = CB >> 0x2;                                /* 移位至指令 cache 位 */
__asm{MRC p15, 0, c3f, c2, c0, 1}                  /* 装载指令 cache 控制寄存器 */
c3f = c3f & ~(0x1 << region);                      /* 清除原来的指令 cache 位 */
c3f = c3f | ((tempCB & 0x1) << region);             /* 设置新的指令 cache 位 */
__asm{MCR p15, 0, c3f, c2, c0, 1}                  /* 保存指令 cache 控制信息 */
}

```

13.2.4 使能区域和 MPU

初始化过程还剩下 2 步：一步是使能活动的区域；另一步是通过使能 MPU、cache 和写缓冲器来启用保护单元硬件。

这里，控制系统再次用到在 13.2.1 小节介绍的 regionSet 例程，以使能一个区域。regionSet 的多种用途在本章最后的例 13.6 有介绍。

为了使能 MPU、cache 和写缓冲器，须修改系统控制寄存器 CP15:c1:c0:0 中的位值。在 ARM940T, ARM946E-S 和 ARM1026EJ-S 处理器中，MPU 位、cache 位和写缓冲器位在 CP15:c1:c0 中的位置是相同的，因此对于这 3 个核，使能配置好的 MPU 是一样的。使能位的位置见图 13.6 和表 13.9。CP15:c1:c0 寄存器还有其它的没有在图 13.6 中表示出的配置位，这些位的目的和位置是与处理器相关的，并不是保护系统的一部分。



图 13.6 CP15:c1:c0 控制寄存器的存储器保护单元控制位

表 13.9 CP15 控制寄存器 1 的保护单元使能位

位	使能的功能	值
0	MPU	0=禁用,1=使能
2	数据 cache	0=禁用,1=使能
12	指令 cache	0=禁用,1=使能

这里使用 `changeControl` 例程,如例 13.5 所述,以使能 MPU 和 cache。`changeControl` 例程可以修改 `CP15:c1:c0:0` 寄存器中的任意位置的值。它下面的 C 函数原型:

```
void controlSet(unsigned value, unsigned mask);
```

传递的第 1 个无符号整型参数包含须改变的位值。第 2 个参数用来选择要改变的位;位值 1 改变控制寄存器的相应位;位值 0 则保持相应位值不变,而不管第 1 个参数的位状态。

例如,若使能 MPU 和 I-cache,禁用 D-cache,则设置位[12]为 1,位[2]为 0,位[0]为 1。第一个参数的值就应为 `0x00001001`,剩下的不改变的位应为 0。为了只选择位[12]、位[2]和位[0]这些要改变的位,将屏蔽码设置为 `0x00001005`。

【例 13.5】 读取控制寄存器,将值保存在一个寄存器中;然后用屏蔽输入值清除所有要改变的位,用 `value` 值设置其为希望的状态;例程最后将新的控制值写到 `CP15:c1:c0` 寄存器。

```
void controlSet(unsigned value, unsigned mask)
{
    unsigned int c1f;

    __asm{MRC p15, 0, c1f, c1, c0, 0 }           /* 读取控制寄存器 */
    c1f = c1f & ~mask;                          /* 清除要改变的位 */
    c1f = c1f | value;                          /* 设置要改变的位 */
    __asm{MCR p15, 0, c1f, c1, c0, 0 }           /* 写控制寄存器 */
}
```

461

13.3 MPU 系统示例

本章前面已经介绍了一系列例程,这些例程可以作为构建块(building blocks)来使用,以初始化和控制一个受保护的系统。本节使用这些例程来初始化和控制一个简单的受保护系统,该系统使用固定的存储器映射。

MPU 系统示例使用本章前面几节介绍的例子,以创建一个实用的受保护系统。它提供了在一个简单的受保护多任务系统中运行 3 个任务的基本框架。这个例子能够清晰地说明 ARM MPU 硬件的概念。该示例用 C 语言编写,使用标准访问权限。

13.3.1 系统需求

本示例系统有以下一些硬件特征:

- 一个带 MPU 的 ARM 核;
- 256 KB 的物理存储器,起始地址为 0x0,结束地址为 0x4000;
- 一些存储器映射的外设,位于 0x10000000~0x12000000 的几兆字节空间里。

在本示例中,所有的存储器映射外设,被看作是一个需要保护的存储器单独空间(参见表 13.10)。

本示例系统有以下一些软件模块:

- 系统软件小于 64 KB。包括向量表、异常处理程序和支持异常的数据堆栈。系统软件必须是用户模式下程序不可访问的,也就是说,用户模式下的任务必须通过系统调用来运行或访问在这个区域中的代码或数据。
- 有一个小于 64 KB 的共享程序,包括通用库和用户任务间传递消息的数据空间。
- 有 3 个在系统中控制独立功能的用户任务。这些任务每个小于 32 KB,当这些任务运行时,它们不能被其它 2 个任务访问。

这些软件被连接,并将软件模块放在分配给它们的区域内。这个示例软件的存储器映射如表 13.10 所列。系统软件的访问权限是系统级的,共享的程序空间可被整个系统访问,任务程序区域包含用户级任务。

表 13.10 受保护示例系统的存储器映射

功 能	访问级别	起始地址	大 小	区 域
保护存储器映射的外围设备	系 统	0x10000000	2 MB	4
受保护的系统	系 统	0x00000000	4 GB	1
共享的系统	用 户	0x00010000	64 KB	2
用户任务 1	用 户	0x00020000	32 KB	3
用户任务 2	用 户	0x00028000	32 KB	3
用户任务 3	用 户	0x00030000	32 KB	3

13.3.2 使用存储器映射分配区域

表 13.10 的最后一列为分配给存储器空间的 4 个区域。使用在表中列出的起始地址、代码、数据块的大小来定义区域。表示区域布局的存储器映射如图 13.7 所示。

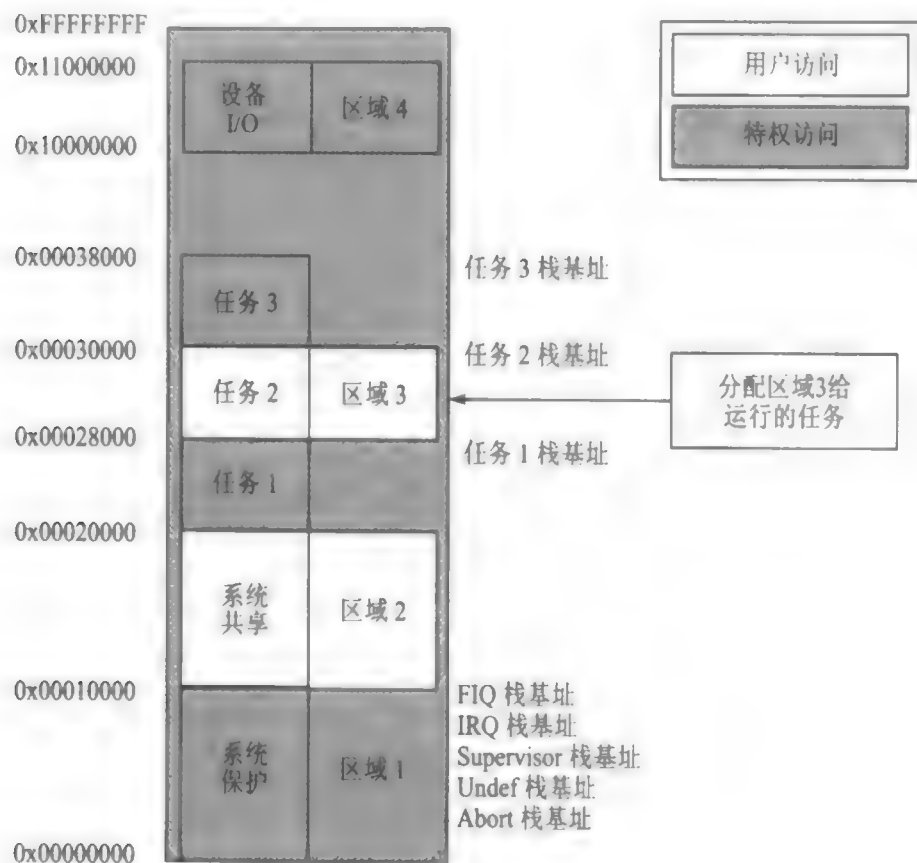


图 13.7 区域分配和受保护示例系统的存储器映射

区域 1 是一个背景区域，覆盖整个可寻址的存储器空间。它是一个特权区域（不允许用户模式下的访问），使能指令 cache，数据 cache 使用直写策略操作。该区域的优先级最低，因为它是最小分配号的区域。

区域 1 的主要功能是限制对 0x0~0x10000 之间的 64 KB 空间的访问，即受保护的系统区域。区域 1 还有另外 2 个功能：作为背景区域和作为睡眠态用户任务的保护区域。作为背景区域，它保证整个存储器空间默认被分配系统级访问，这样就能防止用户任务访问备用的或未用的存储空间；作为用户任务保护区域，它保护睡眠态任务不受运行态任务的非法访问（见图 13.7）。

区域 2 控制对共享系统资源的访问。它的起始地址为 0x10000，大小为 64 KB。它直

接映射在共享系统代码的共享存储器空间上。区域 2 是受保护的区域 1 的一部分,但优先于受保护的区域 1,因为它有较高的区域号。区域 2 允许用户级和系统级的存储器访问。

区域 3 控制运行任务的存储器空间和属性。当控制权从一个任务传给另一个任务时,如上下文切换,操作系统就重新定义区域 3,使得它覆盖运行任务的存储器空间。当区域 3 重新分配到一个新的任务时,它将原来任务的空间交给区域 1 的属性控制。原来的任务空间变成了区域 1 的一部分,而当前运行的任务是一个新的区域 3。这样运行任务就不能访问原来的任务空间,因为它受区域 1 属性的保护。

区域 4 是存储器映射的外围系统空间。该区域的主要目的是,建立没有 cache 和没有写缓冲器的空间。这样,在对控制寄存器和 I/O 设备操作时,就不会涉及到高速缓存可能产生的陈旧数据信息,同时还可以避免使用写缓冲导致的时间或顺序问题(详细的、使用带 cache 和写缓冲器的 I/O 设备的内容见第 12 章)。

13.3.3 初始化 MPU

为了便于组织初始化过程,这里创建了一个称为 Region 的结构。该结构的成员项保存系统操作中使用的区域的属性。在使用 MPU 时,Region 结构并不是必需的,它只是为示例程序更方便设计。在这个示例中,称这些数据结构的集合为一个区域控制块 RCB(Region Control Block)。

初始化程序使用存储在 RCB 中的信息来配置 MPU 中的区域。可以在 RCB 中定义比物理区域更多的 Region 结构。例如,区域 3 是任务使用的惟一物理区域,但是它对应 3 个 Region 结构,每个用户任务使用一个 Region 结构。该结构的定义如下:

```
typedef struct {  
    unsigned int number;  
    unsigned int type;  
    unsigned int baseaddress;  
    unsigned int size;  
    unsigned int IAP;  
    unsigned int DAP;  
    unsigned int CB;  
} Region;
```

Region 结构有 7 个成员项,前 2 个值描述结构本身的特征:分配给 Region 结构的 MPU 区域号 and 使用的访问权限的类型,即 STANDARD(标准类型)或 EXTENDED(扩展类型)。剩下的成员项是所指定区域的属性:区域的起始地址 baseaddress;区域的大小 size;访问权限 IAP 和 DAP;cache 和写缓冲器配置 CB。

示例中 RCB 的 6 个 Region 结构如下:

```
/*          区域号, AP 类型 */
/*          起始地址, 大小, IAP, DAP, CB */

Region peripheralRegion = {PERIPH, STANDARD,
                           0x10000000, SIZE_1M, RONA, RWN, ccb};
Region kernelRegion =    {KERNEL, STANDARD,
                           0x00000000, SIZE_4G, RONA, RWN, CWT};
Region sharedRegion =    {SHARED, STANDARD,
                           0x00010000, SIZE_64K, RORO, RWRW, CWT};
Region task1Region =     {TASK, STANDARD,
                           0x00020000, SIZE_32K, RORO, RWRW, CWT};
Region task2Region =     {TASK, STANDARD,
                           0x00028000, SIZE_32K, RORO, RWRW, CWT};
Region task3Region =     {TASK, STANDARD,
                           0x00030000, SIZE_32K, RORO, RWRW, CWT};
```

访问 RCB 时,为了增强可读性,创建了 13.3.4 小节开关所述的一系列宏。值得注意的是,这里使用一个 4 字母的简单组合,以作为数据存储器 and 指令存储器的访问权限。前 2 个字母表示系统访问权限,后 2 个字母表示用户访问权限。表示系统和用户访问权限的 2 个字母可以是读/写(RW)、只读(RO)或不可访问(NA)。

这里将 cache 和写缓冲器配置信息映射到指令 cache 和数据 cache 的策略属性。第 1 个字母为 C 或 c,分别表示启用或禁用区域的指令 cache。后面 2 个字母表示数据 cache 策略和写缓冲器配置,可以为 WT 或 WB,分别表示直写策略或回写策略。cache 位和写缓冲器位的配置也可以使用字母 c 和 b。Cb 为 WT 的别名,CB 为 WB 的别名,cB 表示禁用 cache 和启用写缓冲器,cb 表示禁用 cache 和禁用写缓冲器。

13.3.4 初始化和配置区域

接下来介绍 configRegion 例程。configRegion 接受 RCB 中的一个 Region 结构指针来配置 CP15 寄存器,使 CP15 寄存器保存有描述区域的数据。

```
/* 区域号分配 */
#define BACKGROUND    0
#define KERNEL        1
#define TASK          2
#define SHARED        3
```

```
#define PERIPH 4
```

```
/* 区域类型分配 */
```

```
#define STANDARD 0
```

```
#define EXTENDED 1
```

```
#define DISABLE 0
```

```
/* 访问权限 */
```

```
#define NANA 0
```

```
#define RWNA 1
```

```
#define RWRO 2
```

```
#define RWRW 3
```

```
#define RONA 5
```

```
#define RORO 6
```

```
/* 区域大小 */
```

```
#define SIZE_4G 31
```

```
#define SIZE_2G 30
```

```
#define SIZE_1G 29
```

```
#define SIZE_512M 28
```

```
#define SIZE_256M 27
```

```
#define SIZE_128M 26
```

```
#define SIZE_64M 25
```

```
#define SIZE_32M 24
```

```
#define SIZE_16M 23
```

```
#define SIZE_8M 22
```

```
#define SIZE_4M 21
```

```
#define SIZE_2M 20
```

```
#define SIZE_1M 19
```

```
#define SIZE_512K 18
```

```
#define SIZE_256K 17
```

```
#define SIZE_128K 16
```

```
#define SIZE_64K 15
```

```
#define SIZE_32K 14
```

```
#define SIZE_16K 13
```

```
#define SIZE_8K 12
```

```
#define SIZE_4K 11
```



```

/* CB= ICache[2], DCache[1], Write Buffer[0] */
/* ICache[2], WB[1;0] = writeback, WT[1;0] = writethrough */
#define CCB      7
#define CWB      7
#define CCBb     6
#define CWT      6
#define CcB      5
#define Ccb      4
#define cCB      3
#define cWB      3
#define cCb      2
#define cWT      2
#define ccB      1
#define ccb      0

/* 区域使能 */
#define R_ENABLE    1
#define R_DISABLE   0

```

例程遵循在 13.2 节列出的初始化步骤。输入参数是一个指向区域 RCB 的指针。例程内部使用 Region 的成员项作为初始化过程的数据输入。该例程的 C 函数原型如下：

```
void configRegion(Region * region);
```

467

【例 13.6】 初始化受保护系统的 MPU、cache 和写缓冲器。

初始化过程中使用了本章前面介绍的例程。这里使用在 13.2 节中列出的步骤来初始化 MPU、cache 和写缓冲器。在例程代码中使用注释标出了每个步骤。执行该例程就初始化了 MPU。

```

void configRegion(Region * region)
{
    /*
    /* 第 1 步——使用 CP15:c6 来定义指令和数据区域的大小和位置 */

    regionSet(region->number, region->baseaddress,
              region->size, R_DISABLE),

    /* 第 2 步——使用 CP15:c5 来设置每个区域的访问权限 */

```

```

        if (region->type == STANDARD)
        {
            regionSetISAP(region->number, region->IAP);
            regionSetDSAP(region->number, region->DAP);
        }
        else if (region->type == EXTENDED)
        {
            regionSetIEAP(region->number, region->IAP);
            regionSetDEAP(region->number, region->DAP);
        }

        /* 第 3 步——分别使用 CP15:c2 和 CP15:c3 来设置每个区域的 cache 和写缓冲器属性 */

        regionSetCB(region->number, region->CB);

        /* 第 4 步——使用 CP15:c6 和 CP15:c1 来使能 cache、写缓冲器和 MPU */

        regionSet(region->number, region->baseaddress,
                  region->size, R_ENABLE);
    }

```

13.3.5 完成初始化 MPU

对于该示例,使用 RCB 来保存描述所有区域的数据,使用最高层的名为 `initActiveRegions` 的例程来初始化 MPU。在系统启动时,该例程被调用一次,为每个活动的区域进行设置。为了完成初始化工作,例程也使能了 MPU。该例程的 C 函数原型如下:

```
void initActiveRegions();
```

该例程没有输入参数。

【例 13.7】 首先为每个在系统启动时处于活跃状态的区域调用一次 `configRegion`: 区域 `kernelRegion`, `sharedRegion`, `peripheralRegion` 和 `task1Region`。

在该示例中,任务 1 是运行的第一个任务。最后调用例程 `controlSet` 来使能 cache 和 MPU。

```

#define ENABLEMPU          (0x1)
#define ENABLEDCACHE      (0x1 << 2)

```

```

#define ENABLEICACHE      (0x1 << 12)
#define MASKMPU           (0x1)
#define MASKDCACHE       (0x1 << 2)
#define MASKICACHE       (0x1 << 12)

void initActiveRegions()
{
    unsigned int value, mask;
    configRegion(&kernelRegion);
    configRegion(&sharedRegion);
    configRegion(&peripheralRegion);
    configRegion(&task1Region);

    value = ENABLEMPU | ENABLEDCACHE | ENABLEICACHE;
    mask = MASKMPU | MASKDCACHE | MASKICACHE;
    controlSet(value, mask);
}

```

13.3.6 受保护系统的上下文切换

至此已经完成了初始化示例系统,控制系统也已经运行了它的第一个任务。在某个时刻,系统可能须做上下文切换,以运行另外一个任务。RCB 中包含当前任务的区域的上下文信息,所以在上下文切换时,无须通过从 CP15 寄存器读取数据来保存区域信息。

为了切换到下一个任务,比如任务 2,操作系统须将区域 3 移动到任务 2 的存储空间上(见图 13.7)。这里再次使用例程 configRegion 来实现这个功能。作为调度的一部分,只需在执行实现当前任务和下一个任务间上下文切换的代码之前调用 configRegion。例程 configRegion 的输入参数是指向 task2Region 的指针,请看下面的汇编代码例子:

```

STMFD    sp!, {r0-r3,r12,lr}
BL       configRegion
LDMFD    sp!, {r0-r3,r12,pc}      ;return

```

等价的 C 调用为:

```
configRegion(&task2Region);
```

13.3.7 mpuSLOS

本章的许多概念和例子代码被包含到一个被称之为 mpuSLOS 的实用控制系统中。

mpuSLOS 是第 11 章介绍的、SLOS 的带存储器保护单元的变体。它实现了与基本的 SLOS 相同的功能,但有以下一些重要的区别:

- mpuSLOS 充分利用了 MPU。
- 应用程序与内核分开,单独编译和构建(build),然后合并到一个二进制文件;连接每个应用程序,使它们在不同的存储空间上执行。
- 每个应用程序被一个称为静态应用程序装载器(static application loader)的例程,装载到不同的固定大小为 32 KB 的区域,装载地址是应用程序的执行地址。因为每个区域大小为 32 KB,所以将堆栈指针设置在 32 KB 的顶端。
- 应用程序只能通过设备驱动程序调用来访问硬件。如果应用程序试图直接访问硬件,那么将产生数据中止异常。这与基本的 SLOS 不同,在基本的 SLOS 中,应用程序直接访问设备,而不产生数据中止异常。
- 跳转到一个应用程序的操作包括:设置 spsr 寄存器,然后使用 MOVS 指令来改变 pc,使它指向任务 1 的入口。
- 每次调用调度程序,活跃的区域 2 就被改变,以反映将要执行的应用程序。

13.4 总 结

存储器保护有两种方法:一种是无硬件保护(无保护)的,这种方法使用强制的软件控制例程来维护任务间相互作用的规则;另一种是有硬件保护(受保护)的,这种方法使用硬件和软件来强制维护任务间相互作用的规则。在受保护的系统里,当访问权限非法时,硬件就会产生一个异常方式来保护存储空间,然后软件作出响应,以处理该异常和管理基于存储器的资源。

ARM 中的 MPU 使用区域作为系统保护的主要概念。区域是一个存储空间属性的集合,也代表一个具有特定属性的逻辑存储空间。区域可以重叠,这样就可以使用背景区域来保护睡眠任务的存储空间不受当前运行任务的非法访问。

初始化 MPU 需要多个步骤,本章介绍了设置各种区域属性的例程。第 1 步是使用 CP15:c6 来设置指令区域和数据区域的大小和位置;第 2 步是使用 CP15:c5 来设置每个区域的访问权限;第 3 步是分别使用 CP15:c2 和 CP15:c3 来设置每个区域的 cache 和写缓冲器属性;最后一步是使用 CP15:c6 来使能活动的区域,并使用 CP15:c1 来使能 cache、写缓

冲器和 MPU。

本章最后介绍了一个简单的多任务环境示例系统。系统有 3 个任务,保护每个任务不受其它 2 个任务的非法访问。示例系统首先定义一个受保护系统,然后显示了如何初始化它。初始化以后,运行一个受保护系统的最后一个步骤:在任务切换时,针对下一个任务改变区域分配。mpuSLOS 是一个包含该示例系统的受保护操作系统的功能性实例。

第 14 章

存储管理单元

- 从 MPU 到 MMU
- 虚存如何工作
- ARM MMU 的详情
- 页 表
- 转换旁路缓冲器
- 域和存储器访问权限
- cache 和写缓冲器
- 协处理器 CP15 和 MMU 配置
- 快速上下文切换扩展
- 示例：一个简单的虚拟存储系统
- MMU SLOS 示例
- 总 结

在创建多任务嵌入式系统时,最好由一个简单的方式来编写、装载及运行各自独立的任务。目前的很多嵌入式系统不再使用自己定制的控制系統,而使用操作系统来简化这个过程。较高级的操作系统采用基于硬件的存储管理单元 MMU。

MMU 提供的一个关键服务是,能使各个任务作为各自独立的程序在其自己的私有存储空间中运行。在带 MMU 的操作系统控制下,运行的任务无须知道其它与之无关的任务的存储需求情况,这就简化了各个任务的设计。

第 13 章介绍了带有存储保护单元 MPU 的处理器核。这些内核只含一个可寻址的物理存储空间,处理器核运行任务时所产生的地址直接用来访问主存。这样,如果 2 个程序编译时使用重叠的地址,那么它们不能同时驻留在主存中。这使得在嵌入式系统中运行多个任务比较困难,因为每个任务都必须运行在主存的不同地址块中。

MMU 简化了任务编程,因为它提供了一些资源,以允许使用虚拟存储器(虚存, *virtual memory*)——一个另外的独立于系统物理存储器的存储空间。MMU 作为转换器,将程序和数据的虚拟地址(编译于虚存中的地址)转换为实际的物理地址,即在物理主存中的地址。这个转换过程允许运行的多个程序使用相同的虚拟地址,而各自存储在物理存储器的不同位置。

从这样 2 个角度看存储器,存储器就有 2 种类型的地址:虚拟地址和物理地址。虚拟地址由编译器和连接器在定位程序时分配;物理地址用来访问实际的主存硬件模块(物理上程序放在这里)。

ARM 公司提供了多种集成有 MMU 硬件的处理器核。这些 MMU 硬件有效地使用虚存来支持多任务环境。本章的目的就是,学习 ARM 存储管理单元的一些基本知识和有关虚存使用的一些基本概念。

本章首先回顾一下 MPU 的保护特性,然后提出一些由 MMU 提供的其它特性。这里将介绍重定位寄存器(relocation register),它保存用来将虚拟地址转换为物理地址的转换数据;将介绍转换旁路缓冲器(TLB),它是一个存放最近地址重定位信息的高速缓存;将介绍如何使用页和页表来配置重定位寄存器。

接下来,要讨论如何通过配置虚存中的页块(blocks of pages)来创建区域。在概述部分的最后,将演示如何使用 MMU 和页表来支持多任务。

将详细说明如何配置 MMU 硬件,分别介绍 ARM MMU 的每个组成部件:页表、转换旁路缓冲器(TLB)、访问权限、cache 和写缓冲器、控制寄存器 CP15:c1,以及快速上下文切换扩展 FCSE(Fast Context Switch Extension)。

本章的最后将提供一个示例程序来演示如何使用虚存建立一个嵌入式系统。示例程序支持 3 个任务运行在多任务环境中,并演示如何保护系统中的每个任务不受其它任务的影响。这些任务被编译运行在同一个虚存执行地址,而放在物理存储器的不同位置。示例的主要部分是演示如何配置 MMU,以将一个任务的虚拟地址转换为物理地址,以及如何在任

务间切换。

本章将示例程序集成到第 11 章介绍的 SLOS 操作系统中,成为 mmuSLOS。

14.1 从 MPU 到 MMU

第 13 章介绍了带有存储保护单元 MPU 的 ARM 核,在那里引入了“区域(region)”的概念,以方便组织和保护存储器。区域可以是活跃的,也可以是睡眠的:活跃区域包含当前系统正在使用的代码或数据;睡眠区域包含当前不使用,但可能在短时间内变为活跃的代码或数据。睡眠区域是被保护的,因此当前正在运行的任务是不能访问它的。

表 14.1 MPU 的区域属性

区域属性	配置选项
类 型	指令、数据
起始地址	大小的倍数
大 小	4 KB~4 GB
访问权限	读、写、执行
cache	回写法、直写法
写缓冲器	使能、禁用

MPU 有专门的硬件来给区域分配属性。可以分配给区域的属性如表 14.1 所列。

本章假设读者已经理解了第 13 章中关于存储保护的概念,因此只简单介绍如何配置 MMU 中的保护硬件。

MPU 和 MMU 的主要区别是,MMU 中增加了额外的硬件,以支持虚存;同时 MMU 硬件将表 14.1 所列的区域属性从 CP15 寄存器移到主存中的表,从而增加了有效区域的数目。

14.2 虚存如何工作

第 13 章介绍了 MPU 并演示了一个多任务嵌入式系统。该系统的每个任务被编译和运行在彼此不同的、固定的主存地址空间,每个任务只能在一个进程空间中运行,任何 2 个任务都不能在主存中有重叠地址。为了运行一个任务,一个保护区被放置在固定地址的程序上,以允许任务访问由该区域定义的一段存储空间。保护区的放置使得该任务得以运行,而其它任务空间被保护。

在 MMU 中,即使任务被编译、连接、运行在主存中有重叠地址的区域中,它们仍然可以运行。MMU 中对虚存的支持可使构建后的嵌入式系统具有多个虚拟存储器映射和单个物理存储器映射。每个任务拥有自己的虚拟存储器映射,以编译和连接组成此任务的代码和数据。内核层管理各个任务在物理存储器中的放置,使得它们在物理存储器中拥有彼此不同的地址,这个地址与其设计时的虚拟运行地址不一样。

为了使任务有各自的虚拟存储器映射,MMU 硬件采用地址重定位(address relocation),即在地址访问主存之前,转换由处理器核输出的存储器地址。可认为在介于内核和主存间的 MMU 中有一个重定位寄存器,这样就能很容易地理解这个转换过程。

当处理器核产生一个虚拟地址时,MMU 取出这个虚拟地址的高位,并用重定位寄存器内的值替换它,从而形成一个物理地址,如图 14.1 所示。

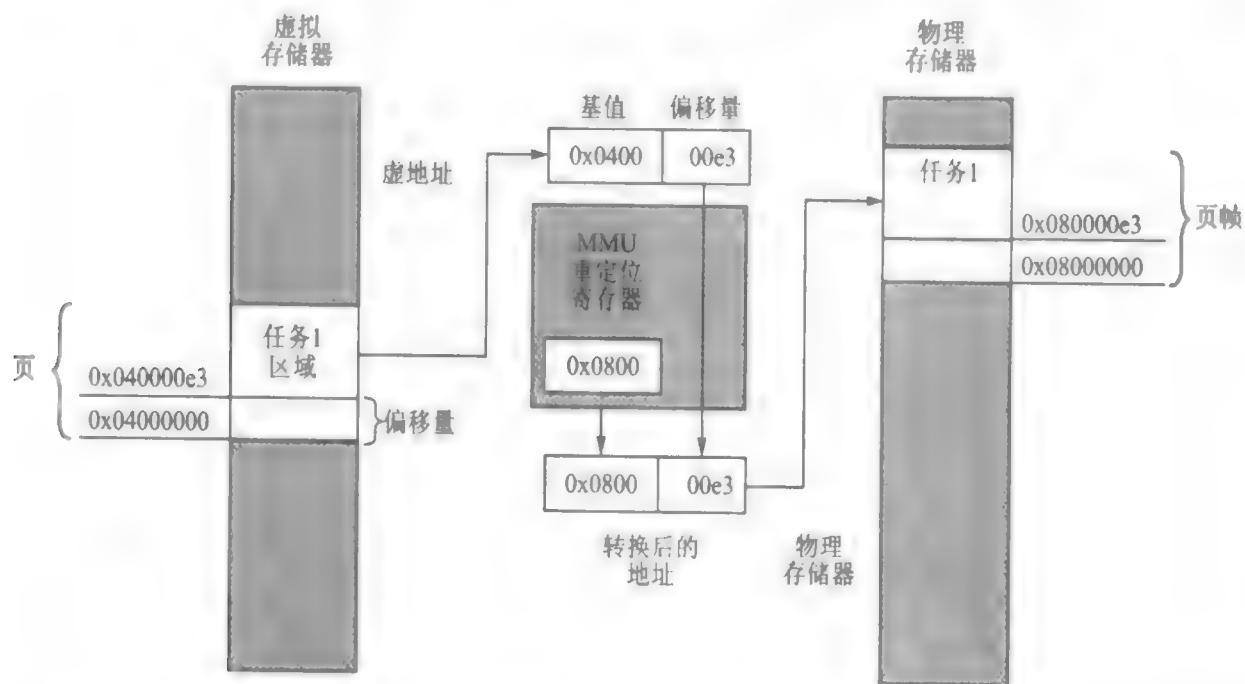


图 14.1 使用重定位寄存器将一个任务从虚拟存储器映射到物理存储器

虚拟地址的低位部分是一个偏移量,它转换成物理存储器的一个特定地址。使用这种方法可以转换的地址范围由这个虚拟地址偏移量部分的最大值所决定。

图 14.1 所示的例子表示一个被编译成以虚拟存储器的 `0x4000000` 为起始运行地址的任务,重定位寄存器将任务 1 的虚拟地址转换成以 `0x8000000` 开始的物理地址。

第 2 个同样被编译成在这个虚拟地址(在这里是 `0x4000000`)运行的任务,可以被放置在任何其它以 `0x10000` (64 KB) 为倍数的地址的物理存储器上,而只需简单地改变一下重定位寄存器的值,就可以将它映射到 `0x4000000` 处。

一个重定位寄存器只能转换一块存储空间。这块存储空间的大小由虚拟地址的偏移量部分所占的位数所决定。这样的一块虚拟存储空间称为一页(page),而转换过程中所对应的那块物理存储空间称为一个页帧(page frame)。

页、MMU 以及页帧之间的关系如图 14.2 所示。ARM MMU 硬件有多个重定位寄存器来支持虚拟地址到物理地址的转换,MMU 需要多个重定位寄存器来有效地支持虚存,因为系统必须将多个页转换到多个页帧。

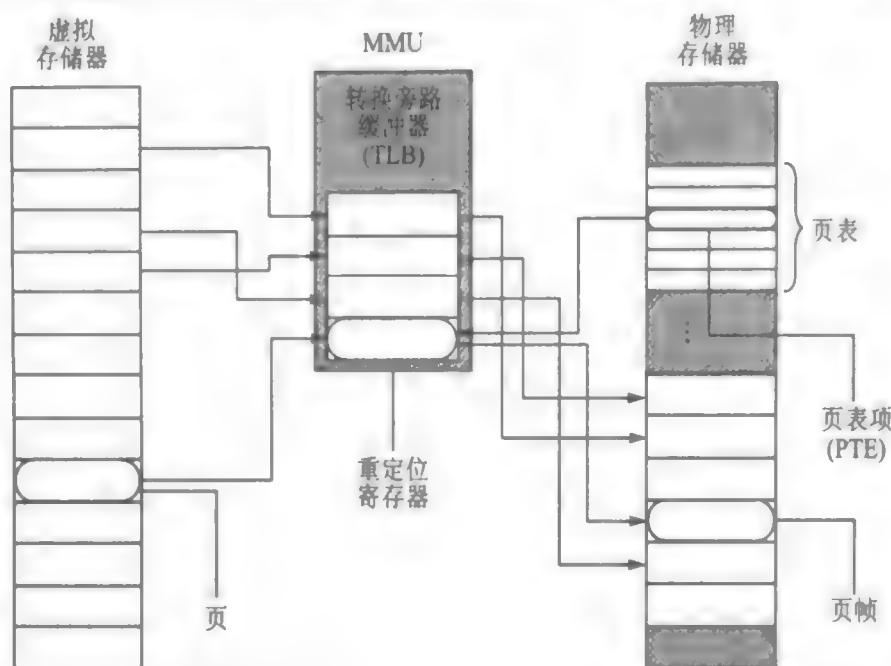


图 14.2 支持虚存系统的组成部件

ARM MMU 中临时存放转换数据的一组重定位寄存器实际上是一个由 64 个重定位寄存器组成的全相联 cache。这个 cache 称为转换旁路缓冲器 (TLB)。TLB 缓存最近被访问的页的转换数据。

除了使用重定位寄存器外,MMU 还使用主存中的表来存放描述系统中用到的虚拟存储器映射的数据,这些转换数据的表称为页表 (page tables)。页表中的每个项代表了将虚拟存储器的一个页转换到物理存储器的一个页帧所需的所有信息。

页表中的每个页表项 PTE (Page Table Entry) 包含关于一个虚拟页的以下一些信息: 用于将虚拟页转换为物理页帧的物理基地址; 分配给该页的访问权限; 页的 cache 和写缓冲器配置。如果参照表 14.1, 会发现 MPU 的大部分区域配置数据现在都保存在页表项中。这意味着, 访问权限、cache 和写缓冲器的行为都以页大小为粒度进行控制, 这在存储器的使用上提供了更好的控制。MMU 中的区域由软件通过将存储器中的虚拟页块进行分组来创建。

14.2.1 使用页定义区域

第 13 章介绍了使用区域来组织和控制用于特殊功能 (比如任务代码和数据, 或存储器输入/输出) 的各块存储空间, 在那里区域被看作是 MPU 体系结构的硬件组成部件。在 MMU 中, 区域被定义为一组页表的集合, 并作为虚存中的连续页完全由软件控制。

虚存中的一页在页表中有一个对应的项,因此虚存的一组连续的页映射到页表的一组连续的项。这样,区域可以被定义为一组连续的页表项的集合。区域的位置和大小可以保存在一个软件数据结构中,而实际的转换数据和属性信息保存在页表中。

图 14.3 所示的例子表示一个任务有 3 个区域:一个用于代码,一个用于数据,第 3 个用于支持任务堆栈。虚存中的每个区域映射到物理存储器的不同块。图中,可执行代码放在 Flash 中,数据和堆栈放在 RAM 中。区域的这种使用方法是支持任务间共享代码的操作系统的典型用法。

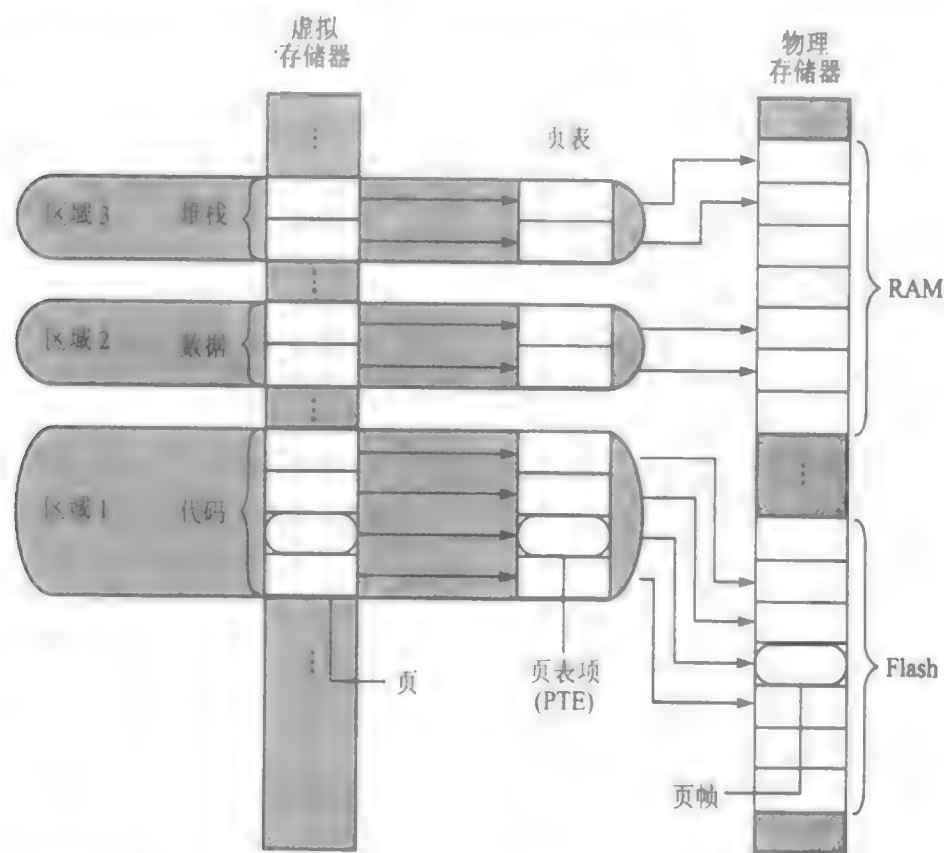


图 14.3 带 MMU 的 ARM 中页映射到页帧的例子

除了一级(L1)页表外,所有其它的页表都代表虚存的 1 MB 空间。如果一个区域的大小大于 1 MB 或者它跨过页表的 1 MB 边界地址,那么就必须用一组页表来描述这个区域。一个区域的页表总是来自 L1 页表的连续页表项,然而在物理存储器中的 L2 页表的位置不需要是连续的。14.4 节将更全面地介绍页表层次。

14.2.2 多任务和 MMU

页表可以驻留在存储器中,而不必映射到 MMU 硬件。构建一个多任务系统的一种方法是,创建几组独立的页表,每组页表对应一个任务的惟一虚存空间。为了激活某个任务,对应这个任务的那组页表和其虚存空间由 MMU 使用,而其它没有被激活的页表则代表睡眠的任务。这种方法使得所有的任务都驻留在物理存储器中。当上下文切换发生以激活一个任务时,这个任务又能立即可用。

通过在上下文切换时激活不同的页表,使得执行有重叠虚拟地址的多个任务成为可能。MMU 可以重定位一个任务的执行地址,而无需在物理存储器中移动这个任务。任务的物理存储空间只是简单地通过激活和不激活页表来映射到虚拟存储空间。图 14.4 所示为 3 个有各自页表集合的任务(它们运行在同一个虚拟地址 0x0400000)的 3 幅图。

图 14.4(a)中,任务 1 正在运行,任务 2 和任务 3 处于睡眠状态;图 14.4(b)中,任务 2 正在运行,任务 1 和任务 3 处于睡眠状态;图 14.4(c)中,任务 3 正在运行,任务 1 和任务 2 处于睡眠状态。每个子图中的虚拟存储器代表正在运行的任务看到的存储空间。3 个子图的物理存储器是一样的,因为它代表真正的物理存储器的实际状态。

图 14.4 还表示出活跃的页表和睡眠的页表,只有正在运行的任务才有活跃的页表集合。睡眠任务的页表驻留在特权物理存储空间中,不能被正在运行的任务所访问。这样就完全保护睡眠任务不受活跃任务的影响,因为从虚拟存储器中不能映射到睡眠任务物理空间。

当页表被激活或不激活时,虚拟地址到物理地址的映射关系也随着改变。因此,每次激活页表后,访问虚存的一个地址可能突然转换成物理存储器的一个不同地址。第 12 章曾经提到,一些 ARM 处理器核有一个逻辑 cache,用来存储(缓存作用)虚拟存储器中的数据。当刚刚提及的地址转换发生时,cache 可能包含无效的、从旧的页表映射来的虚拟数据。为了保证存储器数据的一致性,cache 可能须清除和清理。TLB 可能也须清除,因为它也可能缓存了旧的转换数据。^{*}

虽然清除和清理 cache 及 TLB 会使系统的执行速度有所降低,但是清除和清理 cache 中过时的数据和代码以及 TLB 中过时的转换物理地址,可避免系统因使用无效的数据而发生崩溃。

上下文切换时,并不需要在物理存储器中移动页表数据,而只须改变指向页表位置的指针。

任务间切换需要以下步骤:

^{*} 关于清除和清理 cache 的含义请参阅 12.5 节。——译者注

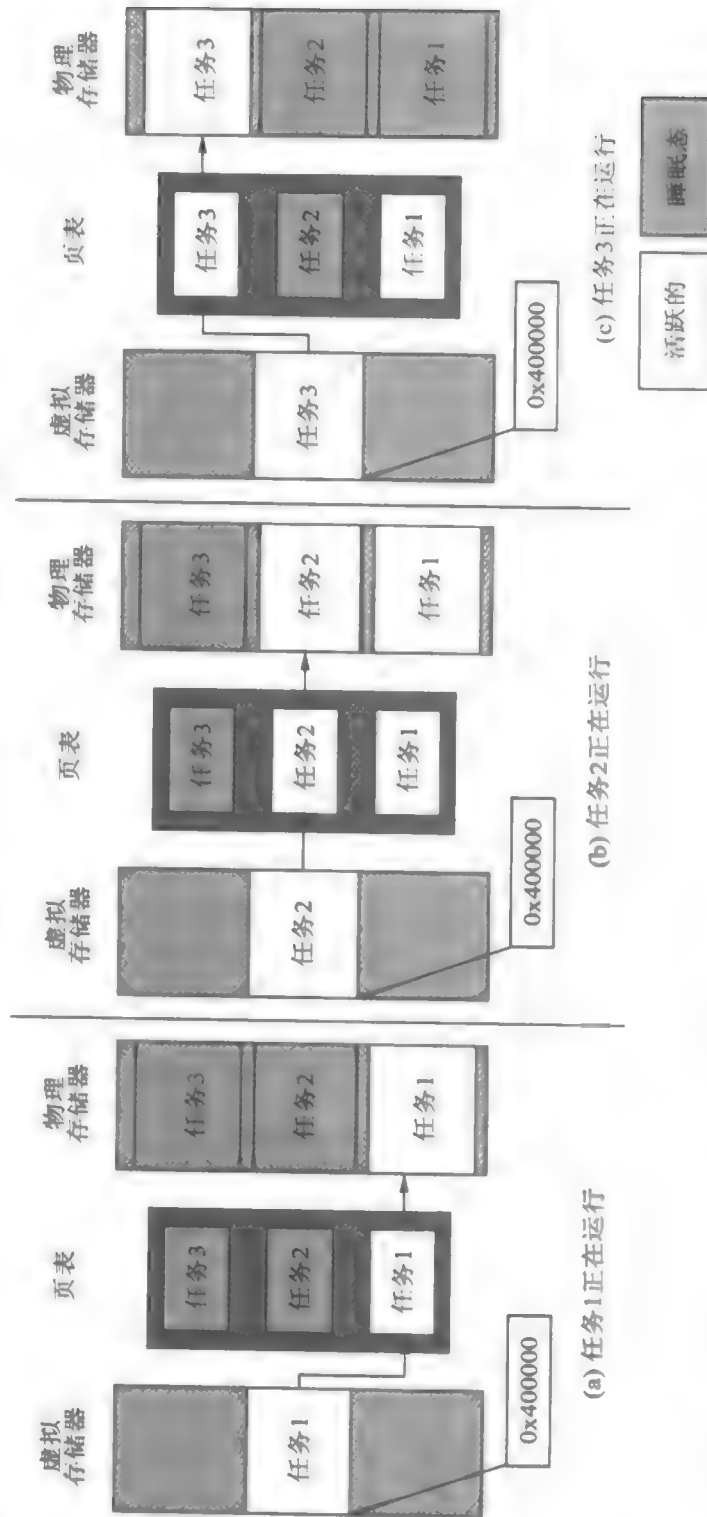


图 14.4 用户任务上下文的虚拟存储器

- ① 保存活跃任务的上下文,并将该任务置为睡眠态;
- ② 清除 cache,如果使用回写策略,则要清理 D-cache;
- ③ 清除 TLB,从而移除原任务的转换数据;
- ④ 配置 MMU,以使用新的页表,把虚拟运行空间转换为被唤醒任务在物理存储器中的位置;
- ⑤ 恢复被唤醒任务的上下文;
- ⑥ 继续执行恢复的任务。

注意:为了缩短执行上下文切换所花费的时间,在 ARM9 系列中可以采用 cache 直写策略。清理数据 cache 需要对 CP15 寄存器写几百次,而将数据 cache 配置为使用直写策略,就不需要在上下文切换时清理数据 cache,从而获得更好的上下文切换性能。使用直写策略,将这些写操作分布在任务的整个操作过程中。虽然回写策略有较好的整体性能,但是使用直写策略,对小型嵌入式系统的代码编写则更简单。

使用这种简化是因为大部分系统使用 Flash 存储器作为非易失性存储介质,而在系统运行时将程序复制到 RAM 中。如果系统含有文件系统并使用动态分页,那么应该采用 cache 回写策略,因为对文件系统存储介质的访问时间通常是访问 RAM 时间的几十倍至上千倍。

如果在性能分析后,发现采用直写策略的效率不高,那么采用 cache 回写策略,性能就能得到改善。如果使用磁盘驱动器或其它非常慢的 2 级存储器(辅存),则采用回写策略应该是没错的。

上面的讨论仅仅适用于使用逻辑 cache 的 ARM 核。如果使用物理 cache,像 ARM11 系列,则当 MMU 改变虚存映射时,cache 中的信息仍然有效。这样,就不需要在改变虚拟存储地址时执行 cache 管理操作。cache 方面的更多知识,请参见第 12 章。

14.2.3 虚存系统的存储器组织

典型的,页表存放在主存的一块空间中,虚拟地址到物理地址的映射是固定的。所谓“固定的”,是指通常操作中,页表中的数据是不会发生变化的,如图 14.5 所示。存储器中的这块固定空间还包含操作系统内核以及其它一些进程。从图 14.5 中可以看到,包含 TLB 的 MMU 是在虚拟或物理存储空间之外操作的硬件,其功能是在 2 个存储空间之间转换地址。

这种固定映射的好处在上下文切换中可以看到。把系统软件放在一个固定的虚拟存储器位置,这样就消除了一些存储器管理任务和流水线影响——如果处理器正在一块虚拟空间上执行,突然这块虚拟空间重新映射到另外一块不同的物理空间引起的。

当在 2 个应用程序任务间实现上下文切换时,处理器其实要发生多次上下文切换。它先从用户模式任务切换到内核模式任务,以处理准备运行下一个应用程序任务时的上下文

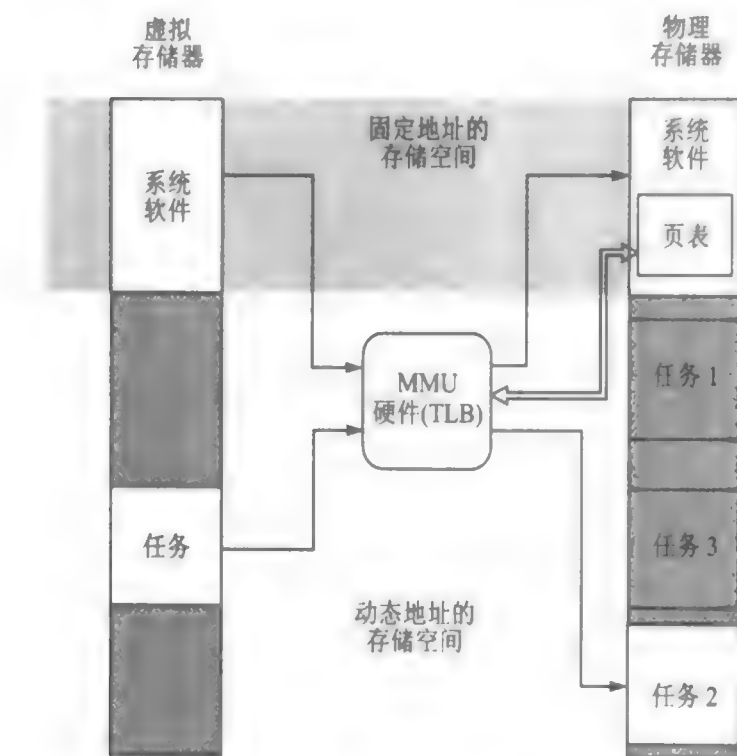


图 14.5 使用 MMU 的系统存储器组织

数据的移动;然后,它从内核模式任务切换到下一个上下文的新的用户模式任务。

通过在虚存的一块固定空间(对所有用户任务都可见的)上共享系统软件,一个系统调用可以直接跳转到这块系统空间,而不必担心需要将页表改为映射到内核进程中。将内核代码和数据映射到所有任务的同一个虚拟地址,避免了需要改变存储器映射,并且避免了需要消耗一个时间片的独立内核进程。

跳转到一个固定的内核空间也避免了流水线结构中固有的问题。处理器核正在一块存储空间内运行代码,如果这块存储空间改变了地址,内核已经从原来的物理存储空间中预取了几条指令,则在新指令从新映射的存储空间中取出并填充流水线时,已经预取的几条指令会被执行。除非特别小心,否则执行原来的存储映射的流水线中的指令可能会破坏程序的正常执行。

这里建议在固定的地址空间(该地址空间的虚拟地址到物理地址的映射是一直不变的)中执行系统代码时激活页表。这种方法保证了用户任务间的安全切换。

许多嵌入式系统不使用复杂的虚存,而只简单地创建一个“固定的”虚拟空间映射来合并所有的物理空间。这些系统通常收集分布在大的地址空间内的多块物理存储块,使其成为一个连续的虚拟存储块。通常它们在初始化过程中创建一个“固定的”映射,该映射在系统操作期间保持不变。

14.3 ARM MMU 的详情

ARM MMU 执行以下一些功能：将虚拟地址转换成物理地址；控制存储访问权限；决定存储器中每一页的 cache 和写缓冲器的行为。当禁用 MMU 时，所有的虚拟地址一一映射到与其相同的物理地址。如果 MMU 在转换一个地址时失败，就会产生一个中止异常。MMU 只有在转换失败、权限错误和域(domain)错误时，才会中止。

MMU 的主要软件配置和控制模块如下：

- 页表；
- 转换旁路缓冲器(TLB)；
- 域和访问权限；
- cache 和写缓冲器；
- CP15:c1 控制寄存器；
- 快速上下文切换扩展。

后续章节将详细介绍这些操作以及如何配置这些模块。

14.4 页 表

ARM MMU 硬件采用 2 级页表结构：一级页表(L1)和二级页表(L2)。

一级页表只有一个 L1 主页表(*L1 master page table*)。L1 主页表包含 2 种类型的页表项：保存指向二级页表起始地址指针的页表项和保存用于转换 1 MB 页的页表项。L1 主页表也称为段页表(*section page table*)。

L1 主页表将 4 GB 的地址空间划分为多个 1 MB 的段(section)，因此 L1 页表包含 4 096 个页表项。L1 主页表是一个混合表，可作为 L2 页表的页目录，也可作为用于转换 1 MB 虚拟页(称为一段)的普通页表。当 L1 页表作为页目录时，其页表项(PTE)包含的是代表 1 MB 虚拟空间的 L2 粗(*coarse*)页表或 L2 细(*fine*)页表的指针；当 L1 页表用于转换一个 1 MB 的段时，其页表项(PTE)包含的是物理存储器中 1 MB 页帧(*page frame*)的首地址。目录页表项和 1 MB 的段页表项可以共存于 L1 主页表。

一个 L2 粗页表有 256 个页表项，占用 1 KB 的主存空间，每个页表项将一个 4 KB 的虚拟存储块转换成一个 4 KB 的物理存储块。粗页表支持 4 KB 或 64 KB 的页，页表项包含的是 4 KB 或 64 KB 的页帧的首地址。如果转换的是一个 64 KB 的页，则对于每个 64 KB 的

页,同一个页表项必须在页表中重复 16 次。

一个 L2 细页表有 1 024 个页表项,占用 4 KB 的主存空间,每个页表项转换一个 1 KB 的存储块。细页表支持 1 KB、4 KB 或 64 KB 虚存页,每个页表项包含 1 KB、4 KB 或 64 KB 物理页帧的首地址。如果转换的是 4 KB 的页,则同一个页表项必须在页表中连续重复 4 次;如果转换的是 64 KB 的页,则同一个页表项需要在页表中连续重复 64 次。

表 14.2 概括了 ARM MMU 中 3 种类型页表的特征。

表 14.2 MMU 中使用的页表

名 称	类 型	页表占用的存储空间/KB	支持的页大小/KB	页表项数目
主页表/段页表	一级(L1)	16	1 024	4 096
细页表	二级(L2)	4	1、4 或 64	1 024
粗页表	二级(L2)	1	4 或 64	256

14.4.1 一级页表项

一级页表支持 4 种类型的页表项:

- 1 MB 段转换项;
- 指向 L2 细页表的目录项;
- 指向 L2 粗页表的目录项;
- 产生中止异常的错误项。

系统通过页表项的最低 2 位[1:0]来确定页表项的类型。页表项的格式要求 L2 页表的地址必须与其页大小的倍数对齐。L1 页表的各种页表项格式如图 14.6 所示。

一个段页表项指向一个 1 MB 的存储段。页表项的高 12 位代替虚地址的高 12 位来产生物理地址。段页表项还包含域属性、cache 属性、缓冲器属性和访问权限属性,这些将在 14.6 节讨论。

粗页表项包含一个 L2 粗页表首地址的指针,同时还包含 L1 表项代表的 1 MB 虚存段的域信息。粗页表必须与 1 KB 的倍数地址对齐。

细页表项包含一个 L2 细页表首地址的指针,同时还包含 L1 表项代表的 1 MB 虚存段的域信息。细页表必须与 4 KB 的倍数地址对齐。

错误页表项产生一个存储页错误。错误条件会导致预取指中止或数据中止,这取决于具体的存储器访问类型。

L1 主页表在存储器中的位置通过写 CP15:c2 寄存器设置。

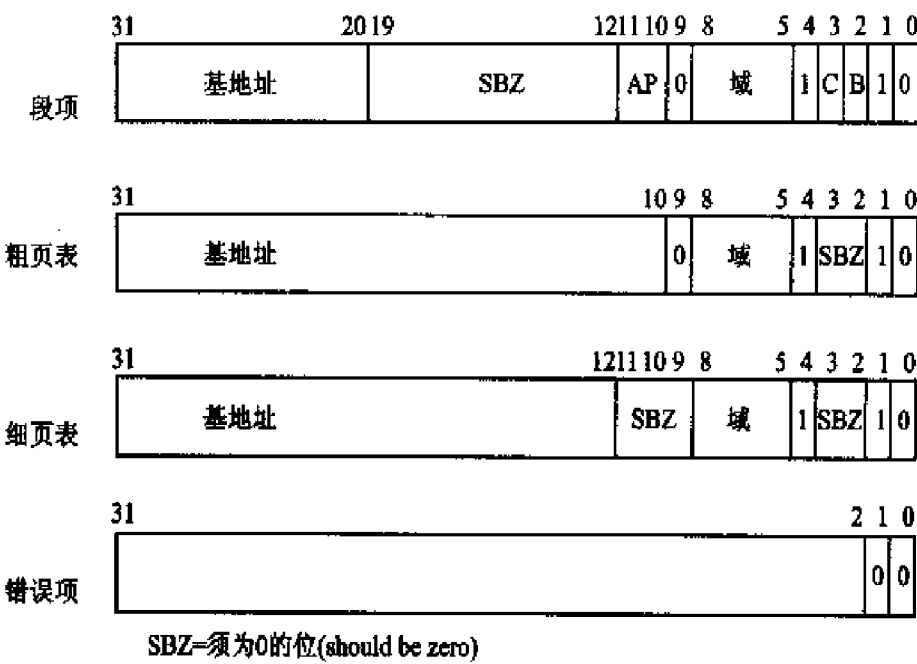


图 14.6 L1 页表项

14.4.2 L1 转换表基地址

CP15:c2 寄存器保存转换表基地址 TTB(Translation Table Base address)——指向 L1 主页表在虚存中的位置。CP15:c2 寄存器的格式如图 14.7 所示。

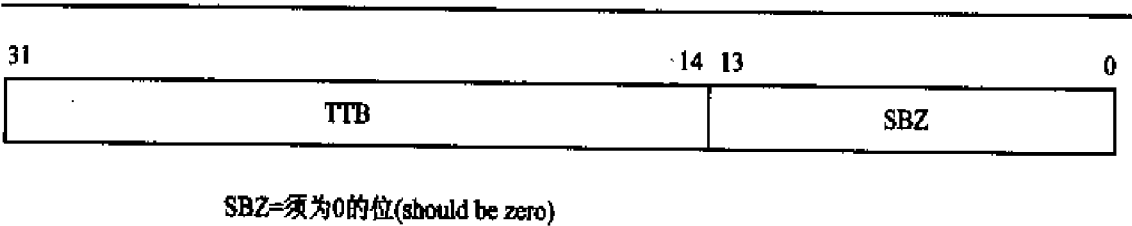


图 14.7 转换表基地址 CP15:c2

【例 14.1】 一个称为 ttbSet 的例程。设置 L1 主页表的 TTB。
ttbSet 例程使用 MRC 指令来写 CP15:c2:c0;0。例程的函数原型如下：

```
void ttbSet(unsigned int ttb);
```

传递给函数的惟一参数是转换表的基地址。TTB 地址必须与存储器的 16 KB 边界对齐。

```
void ttbSet(unsigned int ttb)
{
    ttb &= 0xffffc000;
    __asm(MRC p15, 0, ttb, c2, c0, 0) /* 设置转换表基地址 */
}
```

14.4.3 二级页表项

L2 页表有 4 种可能的页表项(PTE)：

- 定义 64 KB 页帧属性的大(*large*)页表项；
- 定义 4 KB 页帧的小(*small*)页表项；
- 定义 1 KB 页帧的微(*tiny*)页表项；
- 访问时产生页错误中止异常的错误页表项。

L2 页表的页表项格式如图 14.8 所示。MMU 通过页表项的最低 2 位来确定 L2 页表项的类型。

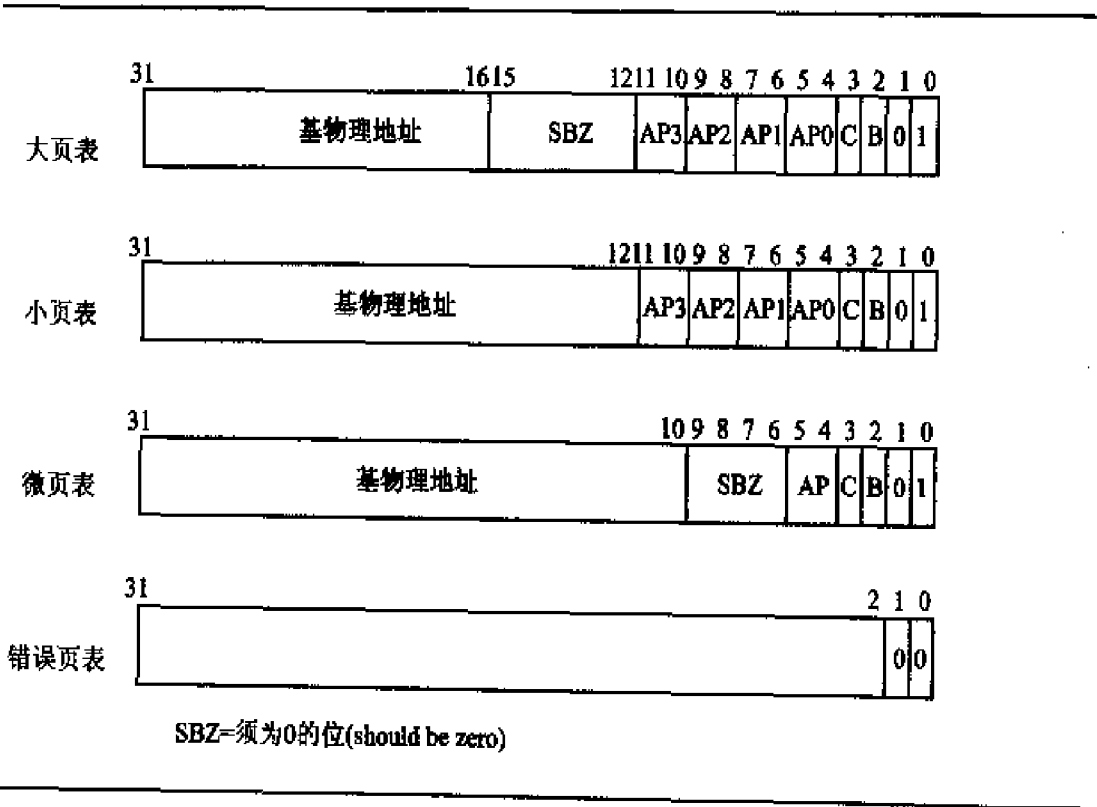


图 14.8 L2 页表项

大页表项包含一个 64 KB 物理存储块的基地址,同时它还含有 4 组权限位域,以及页的 cache 和写缓冲器属性。每一组访问权限位域代表虚存页的 1/4,这些页表项可以看成是 16 KB 子页,以更好地控制 64 KB 页的访问权限。

小页表项保存一个 4 KB 物理存储块的基地址,同时它也含有 4 组权限位域,以及页的 cache 和写缓冲器属性。每一组权限位域代表虚存页的 1/4,这些页表项可以看成是 1 KB 子页,以更好地控制 4 KB 页的访问权限。

微页表项提供一个 1 KB 物理存储块的基地址,同时它含有一个访问权限位域,以及页的 cache 和写缓冲器属性。ARM v6 体系结构不包含微页,如果打算创建一个很容易移植到以后体系结构的系统,则建议在该系统中避免使用 1 KB 微页。

错误页表项产生存储页访问错误。错误条件会导致预取指中止或数据中止,这取决于具体的存储器访问类型。

14.4.4 为嵌入式系统选择合适的页大小

下面是为系统设置页大小的一些技巧和建议:

- 页越小,给定物理存储块就有越多的页帧。
- 页越小,内部碎片会越少。内部碎片是指一页中未使用的存储空间,比如一个大小为 9 KB 的任务可以放在 3 个 4 KB 大小的页中,也可以放在一个 64 KB 大小的页中。使用 4 KB 页,有 3 KB 的未使用空间;而使用 64 KB 页,则有 55 KB 的未使用空间。
- 页越大,系统就越有可能装入引用的代码和数据。
- 二级存储器的访问时间增加时,选择大的页会更有效。
- 随着页的增大,每个 TLB 条目代表存储器中越大的空间,从而系统可以缓存更多的转换数据,将一个任务的所有转换数据装入 TLB 的速度也会更高。
- 如果使用 L2 粗页表,则每个页表占用 1 KB 的存储空间;每个 L2 细页表占用 4 KB 的存储空间。每个 L2 页表转换 1 MB 的地址空间。每个任务使用的最大页表存储空间为:

$$((\text{任务大小}/1 \text{ MB})+1) \times (\text{L2 页表大小}) \quad (14.1)$$

14.5 转换旁路缓冲器

转换旁路缓冲器(TLB)是一个存放最近使用的页转换数据的特殊 cache,它将一个虚拟页映射到一个活跃页帧,并存储用于约束页的访问的控制数据。TLB 是一个 cache,因此有一个丢弃(victim)指针和一个 TLB 行替换策略。在 ARM 处理器核中,当 TLB 失效时,TLB 采用循环算法来选择替换的重定位寄存器。

ARM 处理器核的 TLB 没有很多用来控制其操作的软件命令,TLB 只支持两种类型的命令:清除(flush)TLB 和锁定 TLB 中的转换数据。

存储器访问时,MMU 将虚拟地址的一部分与 TLB 中的所有值进行比较。如果 TLB 中已有所要的转换数据,即为一次 TLB 命中,则由 TLB 提供物理地址转换数据。

如果 TLB 中不存在有效的转换数据,即为一次 TLB 失效,则 MMU 会由硬件自动处理 TLB 失效,通过主存中的页表搜索有效的转换数据,并将其装入 TLB 的 64 行中的一行。在页表中搜索有效的转换数据称为页表搜索(page table walk)。如果找到一个有效的页表项,则由硬件将该转换地址从页表项复制到 TLB 中,并产生访问主存的物理地址;如果最后搜索到页表的错误页表项,则 MMU 硬件产生中止异常。

当 TLB 失效时,在将数据装入 TLB 和产生所需的地址转换之前,MMU 最多搜索 2 个页表。由于 MMU 转换表硬件搜索页表,一次失效的开销通常是 1~2 个主存访问周期,具体的周期数取决于是在哪个页表中找到转换数据的。如果在 L1 主页表中就找到,则为单步页表搜索(single-stage page table walk);如果在 L2 页表中才找到,则为 2 步页表搜索(two-stage page table walk)。

如果 MMU 产生中止异常,则一次 TLB 失效可能会消耗更多的额外周期,因为当中止异常处理程序映射在所请求的虚拟空间时需要额外的周期。ARM720T 只有一个 TLB,因为它采用统一总线体系结构。ARM920T, ARM922T, ARM926EJ-S 和 ARM1026EJ-S 有 2 个 TLB,因为它们采用哈佛总线体系结构:一个 TLB 用于指令转换,一个 TLB 用于数据转换。

14.5.1 单步页表搜索

如果 MMU 搜索的是 1 MB 大小的段页,则硬件能用单步搜索找到所要的页表项,因为 1 MB 的页表项是存放在 L1 主页表里的。

对于 1 MB 段页转换的 L1 页表的页表搜寻如图 14.9 所示。MMU 使用虚地址的基地址部分(位[31:20])来选择 L1 主页表中 4 096 个页表项的一个。如果位[1:0]的值为二进

制的 10, 则此页表项含有一个有效的 1 MB 页可用。页表项中的值被复制到 TLB 中, 把它与虚拟地址的偏移量部分合并来组成物理地址。

如果最低 2 位为 00, 则发生错误; 如果为 01 或 11, 则 MMU 执行 2 步页表搜索。

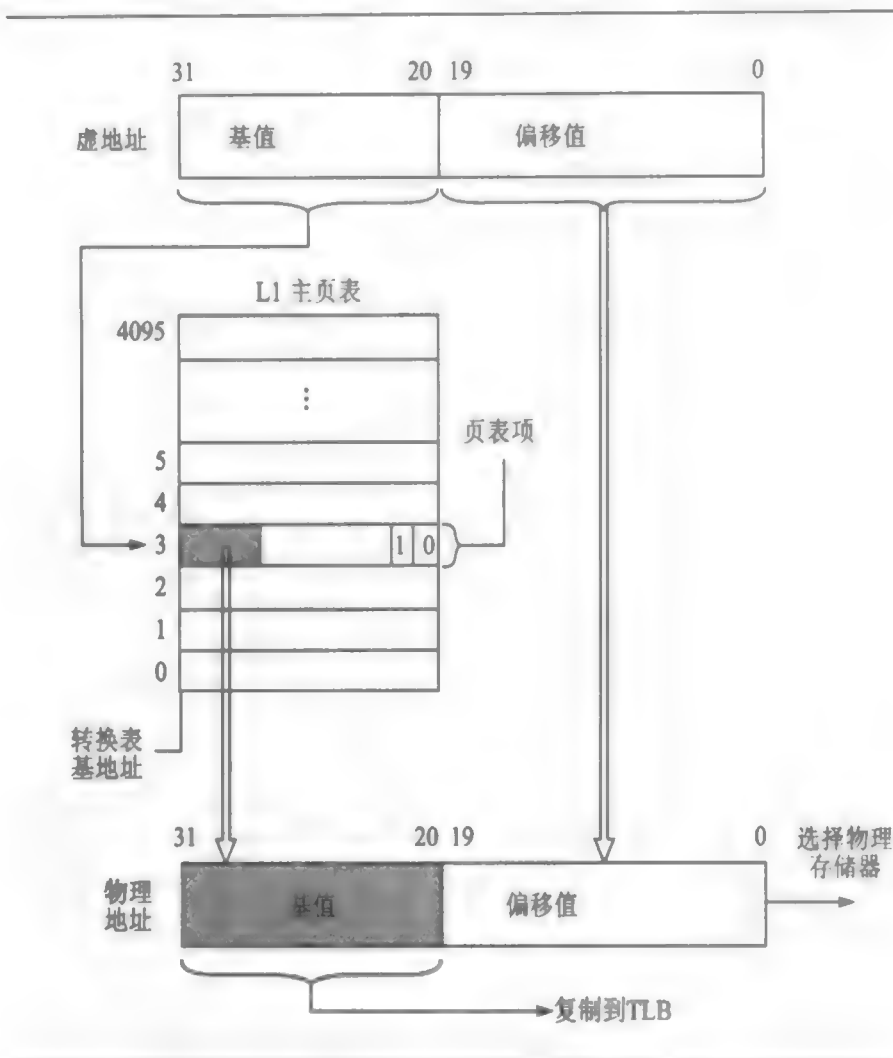


图 14.9 使用 1 MB 段 L1 页表虚实地址转换

14.5.2 2 步页表搜索

如果 MMU 搜索的是大小为 1 KB, 4 KB, 16 KB 或 64 KB 的页, 则页表搜索须执行 2 步才能找到地址转换数据。图 14.10 详细说明了搜索保存在 L2 粗页表的转换数据的过程。这里虚拟地址分成 3 部分。

第 1 步, 用 L1 偏移量部分索引 L1 主页表, 找到虚拟地址的 L1 页表项。如果该页表项的最低 2 位是二进制的 01, 则表示该页表项包含的是一个粗页的 L2 页表基地址(见图 14.6)。

第 2 步, 将 L2 偏移量部分合并到第 1 步找到的 L2 页表基地址, 得到的地址用来选择

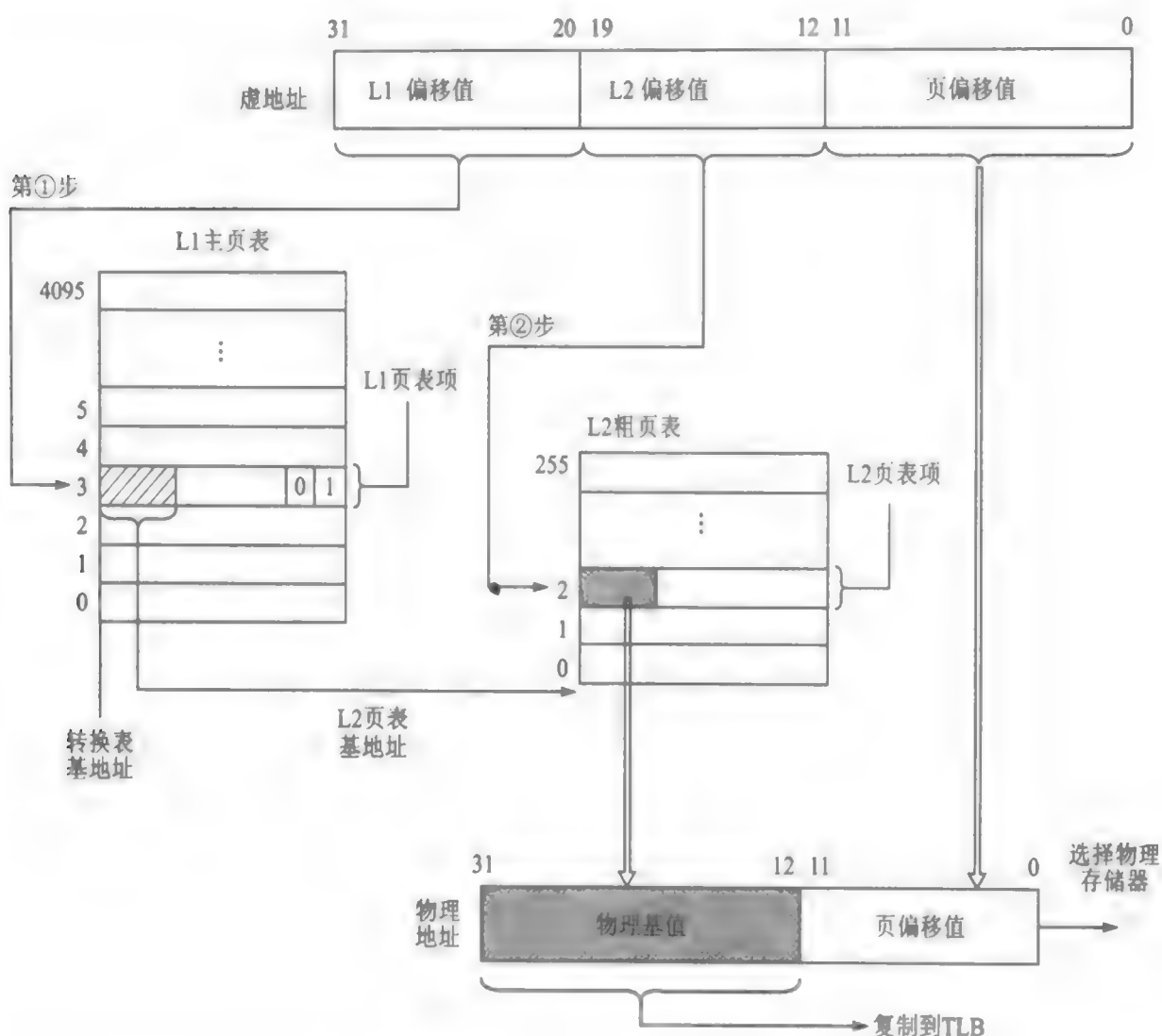


图 14.10 使用粗页表和 4 KB 页的二级虚实地址转换

包含所搜索页的转换数据的页表项。MMU 将 L2 页表项的数据复制到 TLB, 基地址与虚拟地址的偏移量部分合并起来组成所请求的物理存储器地址。

14.5.3 TLB 操作

如果操作系统改变了页表中的数据, 那么缓存在 TLB 中的转换数据可能就不再有效了。处理器核有一些 CP15 命令, 用于清除 TLB, 从而作废 TLB 中的数据。以下是一些可用的命令(见表 14.3): 清除所有 TLB 数据, 清除指令 TLB, 清除数据 TLB, 也可以一次只清除一行 TLB 数据。

表 14.3 清除 TLB 的 CP15:c7 命令

命 令	MCR 指令	Rd 的值	支持的内核
使所有 TLB 无效	MCR p15, 0, Rd, c8, c7, 0	0	ARM720T, ARM920T, ARM922T, ARM926EJ - S, ARM1022E, ARM1026EJ - S, StrongARM, XScale
按行使 TLB 无效	MCR p15, 0, Rd, c8, c7, 1	要使之无效的 虚拟地址	ARM720T
使指令 TLB 无效	MCR p15, 0, Rd, c8, c5, 0	要使之无效的 虚拟地址	ARM920T, ARM922T, ARM926EJ - S, ARM1022E, ARM1026EJ - S, StrongARM, XScale
按行使指令 TLB 无效	MCR p15, 0, Rd, c8, c5, 1	要使之无效的 虚拟地址	ARM920T, ARM922T, ARM926EJ - S, ARM1022E, ARM1026EJ - S, StrongARM, XScale
使数据 TLB 无效	MCR p15, 0, Rd, c8, c6, 0	要使之无效的 虚拟地址	ARM920T, ARM922T, ARM926EJ - S, ARM1022E, ARM1026EJ - S, StrongARM, XScale
按行使数据 TLB 无效	MCR p15, 0, Rd, c8, c6, 1	要使之无效的 虚拟地址	ARM920T, ARM922T, ARM926EJ - S, ARM1022E, ARM1026EJ - S, StrongARM, XScale

【例 14.2】 一个使 TLB 无效的 C 例程：

```
void flushTLB(void)
{
    unsigned int c8format = 0;
    __asm{MCR, p15, 0, c8format, c8, c7, 0}    /* 清除 TLB */
}
```

14.5.4 TLB 锁定

ARM920T, ARM922T, ARM926EJ - S, ARM1022E 和 ARM1026EJ - S 支持 TLB 转换数据的锁定。如果 TLB 中的某一行是锁定的,则当 TLB 清除命令发出时,它仍然保留在 TLB 中。各种 ARM 核的可用锁定命令如表 14.4 所列。用于 TLB 数据锁定的 MCR 指令使用的内核寄存器 Rd 的格式如图 14.11 所示。

表 14.4 访问 TLB 锁定寄存器的命令

命 令	MCR 指令	Rd 的值	支持的内核
读数据 TLB 锁定	MRC p15, 0, Rd, TLB 锁定 c10, c0, 0		ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale
写数据 TLB 锁定	MRC p15, 0, Rd, TLB 锁定 c10, c0, 0		ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale
读指令 TLB 锁定	MRC p15, 0, Rd, TLB 锁定 c10, c0, 1		ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale
写指令 TLB 锁定	MRC p15, 0, Rd, TLB 锁定 c10, c0, 1		ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale

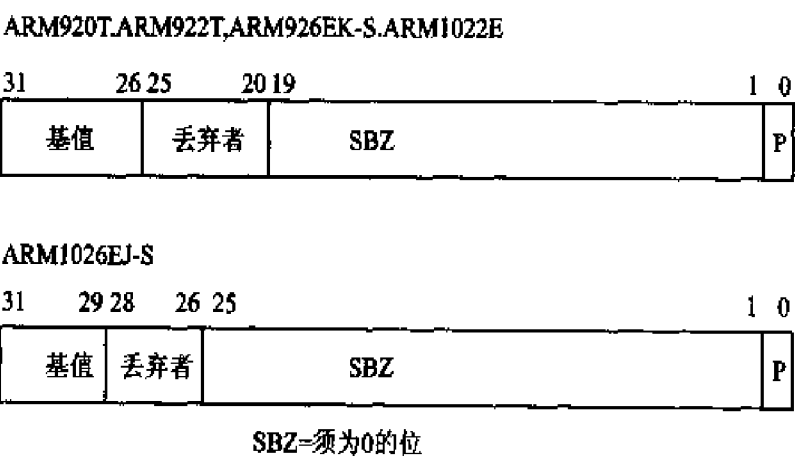


图 14.11 CP15:c10:c0 寄存器的格式

14.6 域和存储器访问权限

有两种不同的控制用来管理一个任务的存储器访问权限:域(domain)用于主控制(primary control);页表中的访问权限用于次控制(secondary control)。

在共享一个共用虚拟存储器映射时,域将一块存储空间与另一块存储空间隔离,以控制虚存的基本访问。有 16 种不同的域可以分配给虚存的 1 MB 段,并通过设置 L1 主页表项(PTE)中的域的有关位(位域)来分配给一个段(见图 14.6)。

当一个域分配给了一个段时,它必须遵守分配给这个域的访问权限。域的访问权限在

CP15;c3 寄存器中分配,它控制处理器核访问虚存段的能力。

16 个可用的域,每个域使用 CP15;c3 寄存器的 2 位来定义访问权限,域访问位取值及对应的意义如表 14.5 所列。图 14.12 给出了 CP15;c3;c0 寄存器的格式,它保存域访问控制信息,在图中 16 个可用的域分别标以 D0~D15。

即使不使用 MMU 提供的虚存功能,仍然可以把这些内核用作简单的存储保护单元:首先将虚拟存储空间直接映射到物理存储空间;然后为每个任务分配一个不同的域;最后使用这些域来保护睡眠任务(通过将它们的域访问设置成“不可访问”)。

表 14.5 域访问位取值

访 问	位域值	说 明
管理者	11	访问不受控制,不产生权限中止
保 留	10	不可预料
客 户	01	访问受页表项中设置的权限值控制
不可访问	00	产生域错误

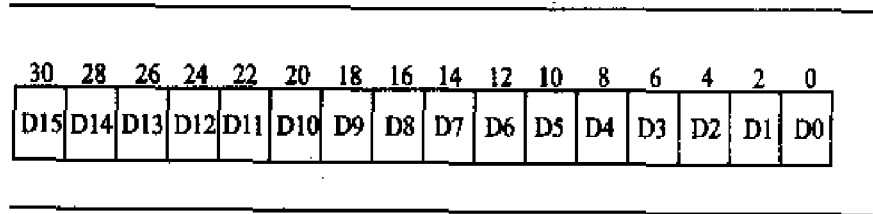


图 14.12 域访问控制寄存器 CP15;c3 的格式

基于页表的访问权限

页表项中的 AP 位决定该页的访问权限,AP 位如图 14.6 和图 14.8 所示。表 14.6 说明了 MMU 如何解释 AP 位域的 2 位值。

表 14.6 访问权限和控制位

特权模式	用户模式	AP 位域	系统位	ROM 位
读/写	读/写	11	忽略	忽略
读/写	只读	10	忽略	忽略
读/写	不可访问	01	忽略	忽略
不可访问	不可访问	00	0	0
只 读	只读	00	0	1
只 读	不可访问	00	1	0
不可预料	不可预料	00	1	1

除了页表项中的 AP 位外,在 CP15;c1 寄存器中还有 2 位起到全局修改存储器访问权限的作用:系统(S)位和ROM(R)位。这 2 位用来在不同模式加速系统中访问大的存储块。

设置 S 位使得所有页具有“不可访问”权限,从而允许特权模式任务对页有读访问权限。因此通过改变 CP15;c1 中的一个位,所有标识为不可访问的空间一下子变为可用,而不需要改变每个页表项的 AP 位域,节省了开销。

改变 R 位使得所有页具有“不可访问”权限,因而特权模式任务和用户模式任务对页都有读访问权限。同样,这一位可以加速对大块存储块的访问,而不需要修改许多页表项的值。

14.7 cache 和写缓冲器

第 12 章介绍了 cache 和写缓冲器的基本操作,这里可以通过页表项中的 2 位(见图 14.6 和图 14.8)来配置存储器中每一页的 cache 和写缓冲器。当配置的是指令页时,写缓冲器位被忽略,cache 位决定 cache 的操作。设置该位,则该页使用 cache;清除该位,则该页不使用 cache。

当配置的是数据页时,写缓冲器位有 2 个用途:使能或禁用页的写缓冲器;设置页的 cache 写策略。页的 cache 位控制写缓冲器位的意义:当 cache 位为 0 时,如果写缓冲器位为 1,则使能写缓冲器,如果写缓冲器位为 0,则禁用写缓冲器;当 cache 位为 1 时,使能写缓冲器,cache 的写策略由写缓冲器位的状态决定,如果写缓冲器位为 0,则页采用直写策略,如果写缓冲器位为 1,则页采用回写策略。表 14.7 列出了 cache 位和写缓冲器位的各种状态及对应的意义。

表 14.7 配置页的 cache 和写缓冲器

指令 cache		数据 cache		
cache 位	页属性	cache 位	写缓冲器位	页属性
0	不使用 cache	0	0	不使用 cache,不使用写缓冲器
1	使用 cache	0	1	不使用 cache,使用写缓冲器
		1	0	使用 cache,直写策略
		1	1	使用 cache,回写策略

14.8 协处理器 CP15 和 MMU 配置

第 12 章介绍过 `changeControl` 例程,例 14.3 再次用到这个例程,用于使能 MMU、cache 和写缓冲器。

控制 MMU 操作的控制寄存器值见表 14.8 和图 14.13。ARM720T,ARM920T 和 ARM926EJ-S 在控制寄存器的相同位置分别有 MMU 使能位[0]和 cache 使能位[2];ARM720T 和 ARM1022E 有写缓冲器使能位[3];ARM920T,ARM922T 和 ARM926EJ-S 使用分离的指令 cache 和数据 cache,因此需要另外一个位(位[12])来使能指令 cache(I-cache)。所有带 MMU 的处理器核都支持将向量表改变到以 0xffff0000 开始的高地址存储器(位[13])。

表 14.8 控制 MMU 操作的控制寄存器 CP15:c1 的位域描述

位	字母指示	使能功能	控 制
0	M	MMU	0=禁用,1=使能
2	C	(数据)cache	0=禁用,1=使能
3	W	写缓冲器	0=禁用,1=使能
8	S	系 统	见表 14.6
9	R	ROM	见表 14.6
12	I	指令 cache	0=禁用,1=使能
13	V	高地址向量表	0=向量表在 0x00000000 1=向量表在 0xFFFF0000

上述 3 类内核使能一个已配置的 MMU 的方法十分相似。为了使能 MMU、cache 和写缓冲器,需要改变控制寄存器的位[12]、位[3]、位[2]和位[0]。

例程 `changeControl` 操作 CP15:c1:c0:0,以改变控制寄存器 c1 的值。例 14.3 给出了一个设置控制寄存器位的简单 C 例程,可以使用下面的函数原型调用它:

```
void controlSet(unsigned int value, unsigned int mask)
```

传递给例程的第 1 个参数是一个无符号的整型(包含要改变的控制值)。第 2 个参数 `mask` 选择要改变的位,`mask` 变量中为 1 的位将把 CP15:c1:c0 寄存器的相应位变为 `value` 输入参数的相同位的值;0 则保持控制寄存器的相应位不变,而不管 `value` 参数的位状态。

【例 14.3】 例程 `controlSet` 设置 CP15:c1 的控制位。

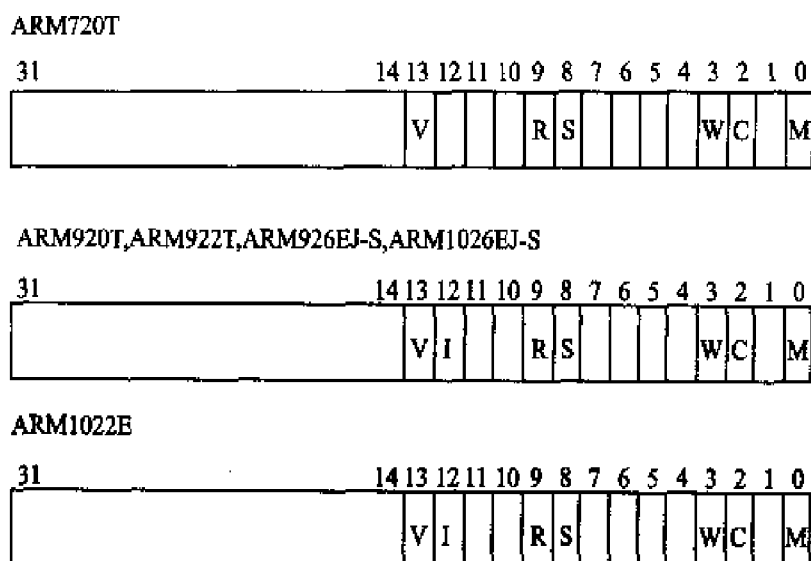


图 14.13 MMU CP15:c1 寄存器控制位

例程首先读取 CP15:c1 寄存器的值,将它放到变量 clformat 中;然后用输入的 mask 值消除 clformat 中需要更新的位,通过将 clformat 和输入参数 value 作“或”操作完成更新;最后将更新后的 clformat 写回到 CP15:c1 寄存器,以使能 MMU、cache 和写缓冲器。

```
void controlSet(unsigned int value, unsigned int mask)
{
    unsigned int clformat;

    __asm(MRC p15, 0, clformat, c1, c0, 0) /* 读取控制寄存器 */
    clformat &= ~mask; /* 清除要改变的位 */
    clformat |= value; /* 设置改变的位 */
    __asm(MCR p15, 0, clformat, c1, c0, 0) /* 写控制寄存器 */
}
```

下面是调用 controlSet 例程使能 ARM920T 的指令 cache、数据 cache 和 MMU 的代码序列;

```
#define ENABLEMMU      0x00000001
#define ENABLEDCACHE   0x00000004
#define ENABLEICACHE   0x00001000

#define CHANGEMMU      0x00000001
#define CHANGEDCACHE   0x00000004
```

ARM 系嵌入式统开发

```
#define CHANGEICACHE      0x00001000

unsigned int enable, change;

#if defined(__TARGET_CPU_ARM920T)
    enable = ENABLEMMU | ENABLEICACHE | ENABLEDCACHE;
    change = CHANGEMMU | CHANGEICACHE | CHANGEDCACHE;
#endif

controlSet(enable, change);
```

14.9 快速上下文切换扩展

快速上下文切换扩展 FCSE(Fast Context Switch Extension)是 MMU 中的一个附加硬件(可以看作是一种增强特征),用于提高 ARM 嵌入式系统的系统性能。FCSE 使得多个独立的任务可以运行在一个固定的重叠存储空间中,而在上下文切换时又不需要清理或清除 cache,或清除 TLB。FCSE 的主要特征是不需要清除 cache 和 TLB。

如果没有 FCSE,则从一个任务切换到另一个任务需要改变虚拟存储映射。如果改变涉及 2 个有重叠地址的任务,则保存在 cache 和 TLB 中的信息将变为无效,这样系统就必须清除 cache 和 TLB。清除这些模块的过程使任务切换增加了很多时间,因为内核不仅要清除 cache 和 TLB 中的无效数据,还要从主存中装载新的数据到 cache 和 TLB。

使用 FCSE,虚拟存储管理增加了一次地址转换。FCSE 在虚地址到达 cache 和 TLB 前,使用一个特殊的、包含进程 ID 值的重定位寄存器来修改虚地址。把第一次转换之前的虚存地址称为虚地址 VA(Virtual Address),把第一次转换之后的地址称为修改后虚地址 MVA(Modified Virtual Address),如图 14.4 所示。当使用 FCSE 时,所有的修改后虚地址都是活跃的,通过使用域访问方式阻止访问睡眠任务,以保护任务。这个问题将在下一节作详细讨论。

这样,任务间的切换就不用涉及到改变页表,只需简单地将新任务的进程 ID 写到位于 CP15 的 FCSE 进程 ID 寄存器。正是因为任务切换不需要改变页表,因而切换后 cache 和 TLB 中的值依然保持有效,不再需要清除。

当使用 FCSE 时,每个任务必须在 0x00000000~0x1FFFFFFF 的固定虚存地址范围内执行,且必须位于修改后虚存的不同的 32 MB 空间中。系统共享 0x2000000 上面的所有存储空间,使用域来实现任务间保护。正在运行的任务通过其当前进程 ID 进行识别。

为了利用 FCSE,编译连接所有的任务,使它们都运行在虚存的第一个 32 MB 块空间,为每个任务分配一个进程 ID;然后使用下面的重定位公式,将每个任务放置在修改后虚存的不同 32 MB 分区(partition)中:

$$MVA = VA + (0x2000000 * \text{进程 ID}) \quad (14.2)$$

为了计算任务在修改后虚存的起始地址,在式(14.2)中,VA 和任务的进程 ID 取 0 即可。

保存在 CP15:c13:c0 寄存器中的值包含当前的进程 ID,寄存器中进程 ID 位域为 7 位宽度,因此可以有 128 个进程 ID。寄存器的格式如图 14.14 所示。

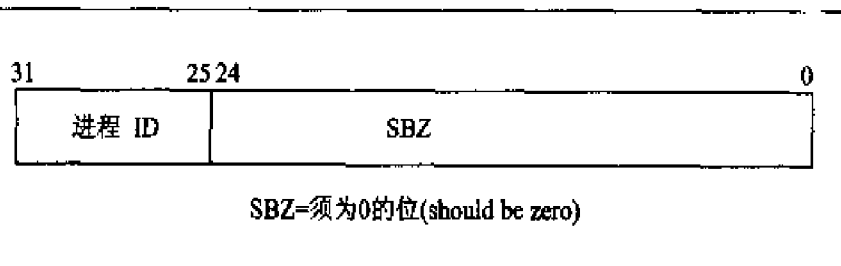


图 14.14 快速上下文切换寄存器 CP15:c13

例 14.4 为一个简单的例程 processIDSet,用来设置 FCSE 中的进程 ID。使用下面的函数原型调用它:

```
void processIDSet(unsigned value);
```

【例 14.4】 以一个无符号整数作为输入,取出低 7 位(模 128)乘以 0x2000000(32 MB),然后使用 MCR 指令将结果写入进程 ID 寄存器。

```
void processIDSet(unsigned int value)
{
    unsigned int PID;
    PID = value << 25;
    __asm{MCR p15, 0, PID, c13, c0, 0} /* 写进程 ID 寄存器 */
}
```

497

14.9.1 FCSE 如何使用页表和域

为了有效地使用 FCSE,系统使用页表来控制区域的配置和操作,使用域来隔离各个任务。请参考图 14.15,图中表示了从任务 1 切换到任务 2 之前和之后的存储器布局。表 14.9 是建立图 14.15 的详细数据。

图 14.16 显示了 CP15:c3:c0 的域访问寄存器值的改变情况(用于实现从任务 1 到任务 2 的切换)。任务间的切换需要更改进程 ID,相应的,域访问寄存器也要被赋予新的内容。

表 14.9 表示域 1 分配给任务 1,域 2 分配给任务 2。当从任务 1 切换到任务 2 时,须改变域访问寄存器,以允许客户(client)访问域 2,而不允许访问域 1,这样就能防止任务 2 访问任务 1 的存储空间。

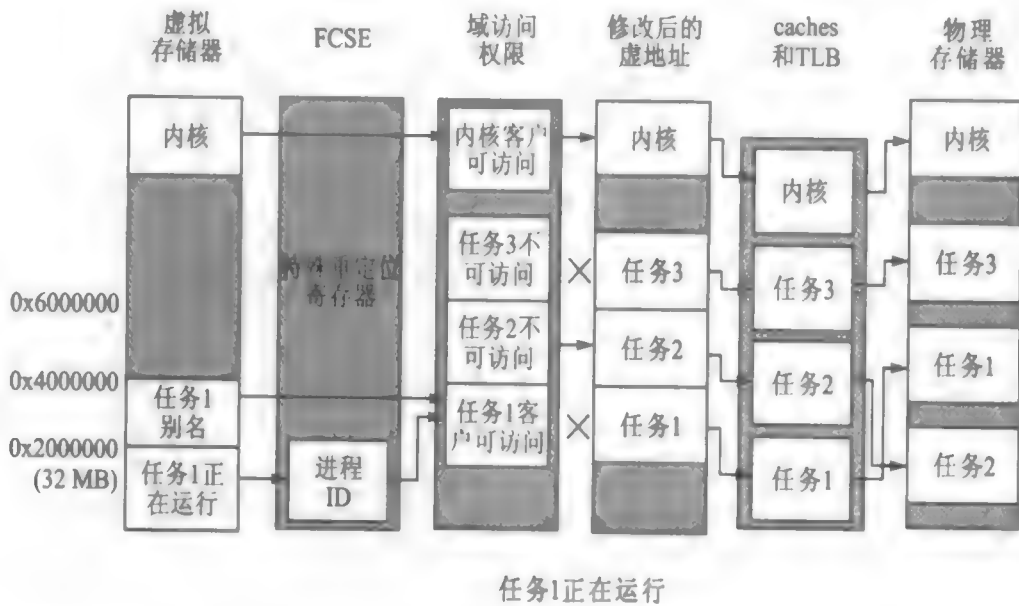
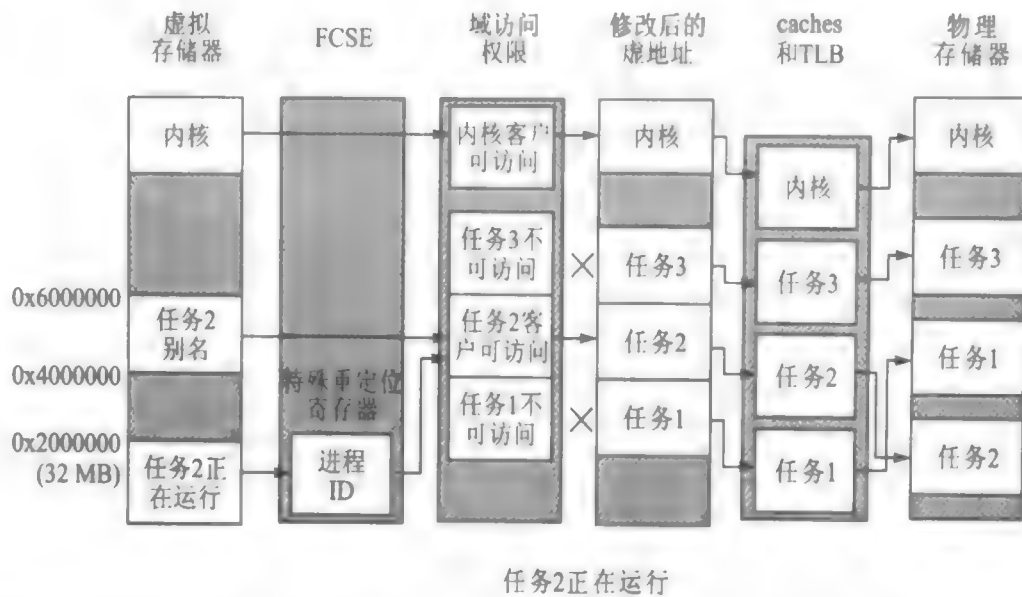


图 14.15 快速上下文切换扩展例子:在一个 3 任务的多任务环境中,
切换前任务 1 运行,切换后任务 2 运行

注意:对于内核,客户访问是保持不变的(域 0)。这允许页表控制存储器的系统空间访问。

在任务间共享存储空间可以通过使用一个“共享”的域来完成,见图 14.16 和表 14.9 中的域 15。共享的域在图 14.14 中没有表示出。任务可以共享一个域,这个域允许客户访问修改后的虚存中的一个分区。2 个任务都可以看到这个共享的域,而访问它要通过对应存

储空间的页表项来控制。

Pre		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
任务1 正在运行		01	00	00	00	00	00	00	00	00	00	00	00	00	00	01	01
Post		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
任务2 正在运行		01	00	00	00	00	00	00	00	00	00	00	00	00	01	00	01

图 14.16 3 任务的多任务环境中,从任务 1 到任务 2 改变前后 CP15 寄存器 3 的值

表 14.9 在一个简单的 3 任务的多任务环境中使用 FSCE 的域分配

区 域	域	特权 AP	用户 AP	修改后虚存中 分块的起始地址	进程 ID
内 核	0	读/写	不可访问	0xFE000000	不分配
任务 3	3	读/写	读/写	0x06000000	0x03
任务 2	2	读/写	读/写	0x04000000	0x02
任务 1	1	读/写	读/写	0x02000000	0x01
共 享	15	读/写	读/写	0xF8000000	不分配

使用 FCSE 时执行一次上下文切换需要以下步骤:

- ① 保存活跃任务的上下文,并将活跃任务置为睡眠态;
- ② 将唤醒任务的进程 ID 写到 CP15:c13:c0 中;
- ③ 通过写 CP15:c3:c0,将当前任务的域设置为不可访问,而唤醒任务的域设置为客户可访问;
- ④ 恢复唤醒任务的上下文;
- ⑤ 继续执行被恢复的任务。

14.9.2 使用 FCSE 的提示

- 任务在大小上有固定的最大 32 MB 的限制。
- 存储器管理者必须使用有固定起始地址(32 MB 的倍数)的固定 32 MB 分区。

ARM 系嵌入式统开发

- 除非想为每个任务管理一个异常向量表,否则使用 CP15 寄存器 1 的 V 位将异常向量表放置在虚地址 0xffff0000。
- 必须定义和使用一个活跃的域控制系统。
- 如果执行发生在第一个 32 MB 块中,则紧随着进程 ID 的改变,内核会从先前的进程空间中取 2 条指令。因此,从存储器中一个“固定”区域切换任务是明智的。
- 如果使用域来控制任务的访问,则正在运行的任务也作为一个别名出现在虚存的 $VA + (0x20000000 * \text{进程 ID})$ 地址。
- 如果使用域来保护各个任务,则除非修改一级页表中域的相应位,并在上下文切换时清除 TLB,否则最多只能有 16 个并发任务。

14.10 示例:一个简单的虚拟存储系统

下面是一个简单的示例,用于说明使用虚存的小型嵌入式系统的基本原理(fundamentals)。它设计成运行在 ARM720T 或 ARM920T 核上。该示例提供一个静态的多任务系统,说明了运行 3 个并发任务所需的基础部分(infrastructure)。它选用的开发工具是 ARM ADS1.2。该示例的主要目的是,理解 ARM MMU 的底层(underlying)硬件原理(尽管还有许多方法可以用来改进该示例)。该示例没有涉及分页和二级存储器交换。

该示例中所有的用户任务使用相同的执行区域,从而简化了这些任务的编译和连接工作。每个任务被编译成一个独立的程序,包含单个区域中的代码、数据和堆栈信息。

硬件需求是一块基于 ARM 的评估板,包含一个 ARM720T 或 ARM920T 的处理器核。例子需要 256 KB 的 RAM,起始地址为 0x00000000;还需要一种将代码和数据装载到存储器的方法;其外还要有若干存储器映射的外设,分布在 0x10000000~0x20000000 的 256 MB 地址范围。

软件需求是一个操作系统的基础部分,比如在前面章节中介绍的 SLOS,系统必须支持固定分区的多任务。

本例使用了 1 MB 和 4 KB 的页(实际上它支持所有的页大小),因为任务的大小限制在小于 1 MB,可放在(fit)单个 L2 页表中,因此只须通过改变主 L1 页表的单个 L2 页表项(PTE)就能执行任务切换。

与另一种方法——为每个任务创建和维护一个完整的页表集合,在切换时改变 TTB 地址相比,这种方法要简单得多。通过改变 TTB 来实现任务存储器映射的改变,需要在 3 个不同的页表集合中创建一个主表和所有的 L2 系统表,这样就需要额外的存储器来保存这些增加的页表。在单个 L2 表交换的目的是,避免在多个页表组中复制系统信息,减少复制的页表数目,也就减少了运行系统所需的存储器。

该示例使用下列 7 个步骤来设置 MMU:

- ① 定义固定的系统软件区域,这个固定的空间如图 14.5 所示。
- ② 为 3 个任务定义 3 个虚存映射,这些映射的总体布局如图 14.4 所示。
- ③ 将步骤①和步骤②列出的区域定位到物理存储器映射,相应结果如图 14.5 的右边部分所示。
- ④ 定义并在页表区域中定位页表。
- ⑤ 定义创建和管理区域与页表的数据结构,这些结构与实现有关,这里是特别为这个例子定义的。然而这些结构的一般形式对大多数简单系统而言,也是很好的起始参考。
- ⑥ 初始化 MMU、cache 和写缓冲器。
- ⑦ 建立上下文切换例程,以顺利地从一个任务转换到下一个任务。

下面各节将详细介绍这些步骤。

14.10.1 第 1 步:定义固定的系统软件区域

这个操作系统使用 4 个固定的系统软件区域:位于 0x00000 的专用 32 KB 内核区域;位于 0x80000 的 32 KB 共享存储区域;位于 0x10000 的专用 32 KB 页表区域;位于 0x10000000 的 256 MB 外设区域(参见图 14.17)。在初始化过程中定义这些区域,以后不再改变其页表。

特权内核区域保存系统软件,包含操作系统内核代码和数据。通过在这个区域使用固定的地址,可以避免改变系统模式时重映射的复杂性。该区域还包含向量表和处理 FIQ,IRQ,SWI,UND 和 ABT 异常的堆栈。

共享存储区域在虚存的一个固定地址上,所有的任务使用这个区域来访问共享的系统资源。共享存储区域包含共享库,以及在上下文切换时从特权模式切换到用户模式所使用的转换例程。

页表区域包含 5 个页表,虽然页表区域大小为 32 KB,但是系统只使用其中的



图 14.17 虚存中的固定区域

ARM 系嵌入式统开发

20 KB: 16 KB 用于主表(master table), 4 KB 用于 L2 表(4 个 L2 表每个使用 1 KB)。

外设区域控制系统设备 I/O 空间, 这个区域的主要目的是, 将该区域配置成无 cache 且无写缓冲的区域。这样, 对输入、输出和控制寄存器来说, 就可以避免因缓存操作而可能包含的陈旧数据; 同时, 也可以避免使用写缓冲器所引起的延时。

这个区域也防止用户模式下访问外围设备, 因此访问外设必须通过设备驱动程序完成。这个区域只允许特权访问, 不允许用户访问。示例中只有单个外设区域, 但是在一个较完整的系统中, 可能需要定义更多的这种区域, 以对各个设备提供更好的控制。

14.10.2 第 2 步: 为每个任务定义虚存映射

3 个用户任务分别在 3 个时间片内运行, 每个任务的虚存映射都是一样的。

每个任务在其存储器映射中可以看到 2 个区域: 一个是位于 0x4000000 的专用 32 KB 任务区域; 另一个是位于 0x8000 的 32 KB 共享存储区域(见图 14.18)。

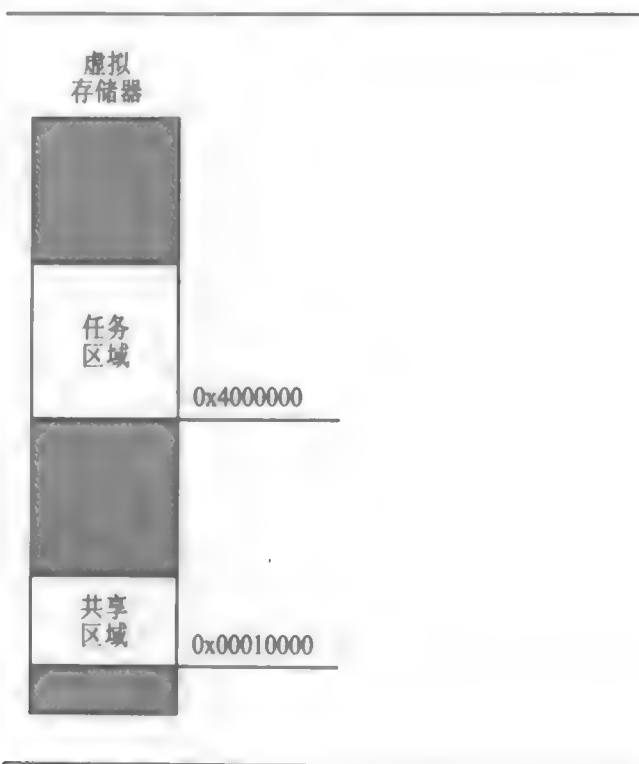


图 14.18 运行的任务所看到的虚拟存储器

任务区域包含运行的用户任务的代码、数据和堆栈。当调度器使控制权从一个任务切换到另一个任务时, 它必须重新映射任务区域, 这是通过改变 L1 页表项, 使它指向即将运行任务的 L2 页表来完成的。改变页表项后, 任务区域就指向下一个要运行的任务的物理地址。

共享区域是一个固定的系统软件区域, 其功能在 14.10.1 小节中已经描述过了。

14.10.3 第3步:在物理存储器中定位区域

示例中定义的区域必须定位在物理存储器的互不重叠或冲突的地址上。所有区域在物理存储器上的定位及其虚地址和大小如表 14.10 所列。表中也列出了为每个区域所选择的页大小和每个区域所需要的页的数目。

表 14.10 MMU 例子中区域的放置

区 域	地 址	区域大小	虚存基地址	页的大小	页的数目	物理基地址
内 核	固 定	64 KB	0x00000000	4 KB	16	0x00000000
共 享	固 定	32 KB	0x00010000	4 KB	8	0x00010000
页 表	固 定	32 KB	0x00018000	4 KB	8	0x00018000
外 设	固 定	256 MB	0x10000000	1 MB	256	0x10000000
任务 1	动 态	32 KB	0x00400000	4 KB	8	0x00020000
任务 2	动 态	32 KB	0x00400000	4 KB	8	0x00028000
任务 3	动 态	32 KB	0x00400000	4 KB	8	0x00030000

表 14.10 列出了 4 个在系统操作过程中使用固定页表的区域:内核、共享存储器、页表和外设区域。

任务区域在系统操作过程中动态改变其页表,针对正在运行的不同任务,它将同一个虚地址转换成不同的物理地址。

图 14.19 显示了各个区域在虚存和物理存储器中的放置情况。内核、共享和页表区域直接映射到物理存储器的连续页帧块,在这个空间的上面,是分配给 3 个用户任务的页帧。物理存储器中的任务是 32 KB 的固定分区,也是连续的页帧。存储器映射的外围 I/O 设备稀疏分布在物理存储器的 256 MB 空间内。

14.10.4 第4步:定义和定位页表

系统中专门用一个区域来预先保存页表,下一步就是将区域内的实际页表定位到物理存储器中。图 14.20 详细表示了页表区域映射到物理存储器的哪个位置。它是图 14.19 中页表的放大,将存储器展开来描述 L1 主页表和 4 个 L2 页表之间的关系。图中还描述了转换数据如何保存在页表中。

一个 L1 主页表定位 L2 页表,并转换外设区域的 1 MB 段。系统 L2 页表包含 3 个系统区域(内核区域、共享存储区域和页表区域)的转换地址数据。有 3 个任务 L2 页表,用来映

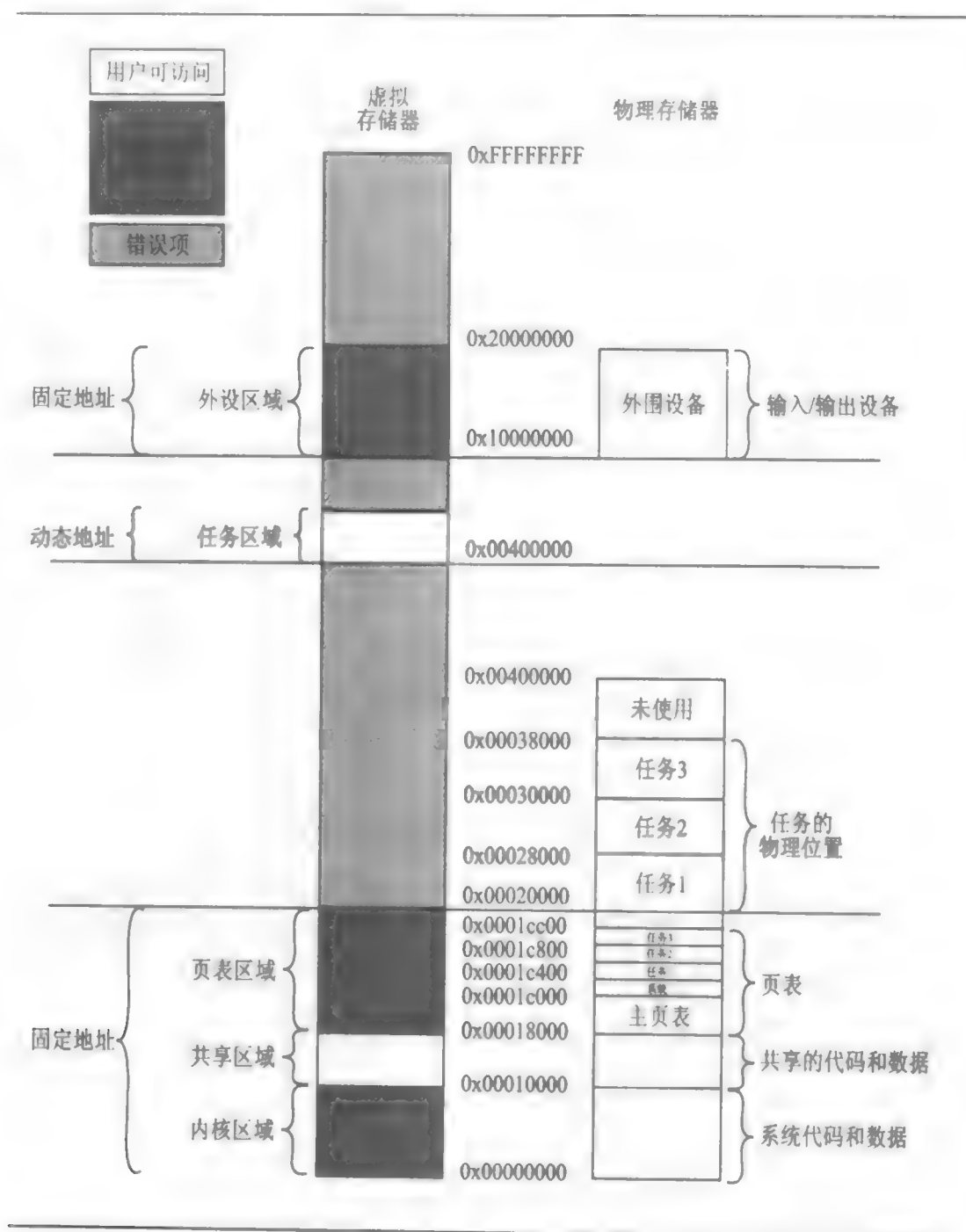


图 14.19 简单虚存例子的存储器映射

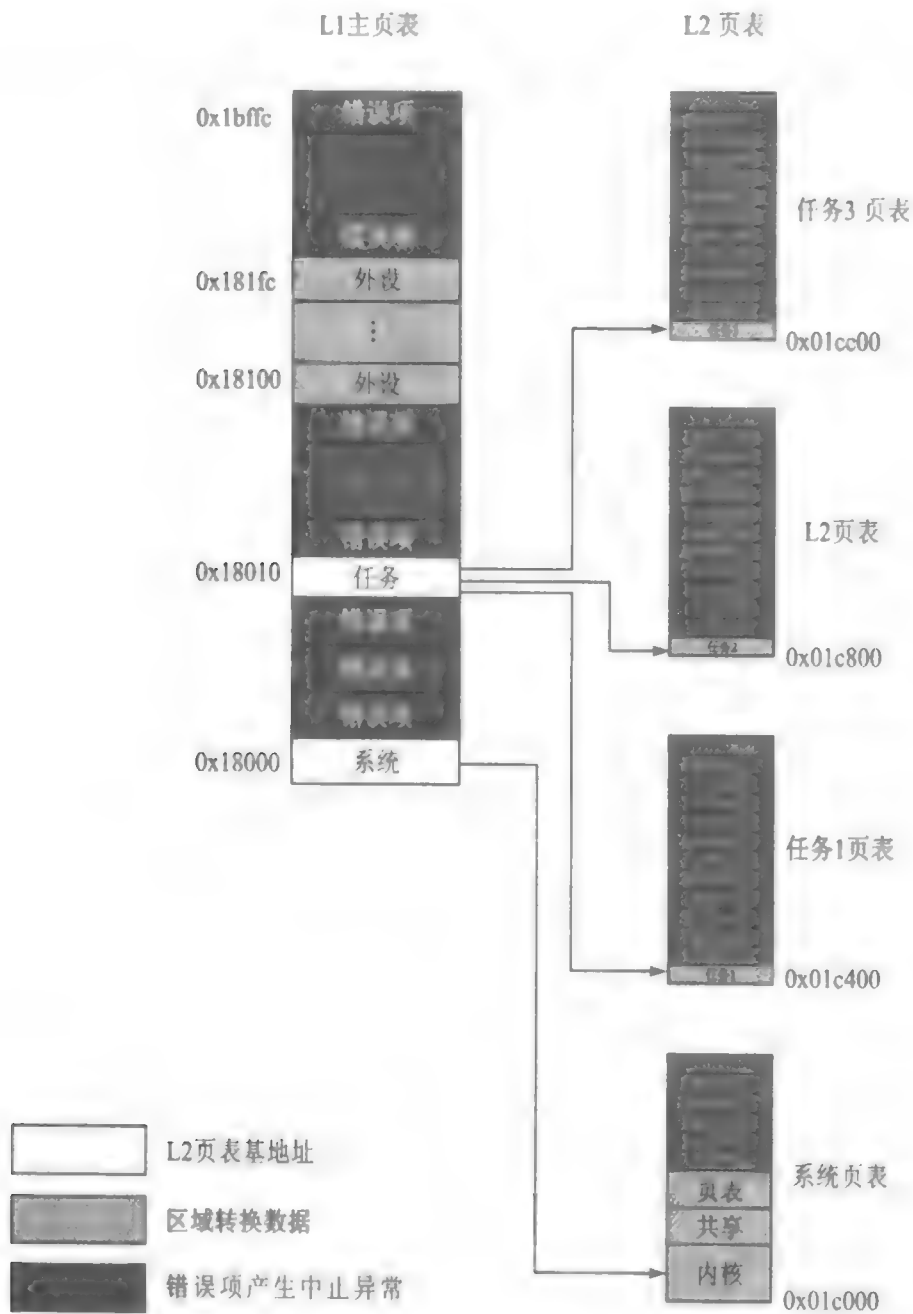


图 14.20 简单虚拟存储器示例中的页表结构

射到 3 个并发任务的物理地址。

5 个页表中只有 3 个在运行时是同时有效的：L1 主页表、L2 系统页表和 3 个 L2 任务页表中的一个。

调度器通过上下文切换重新映射任务区域，控制哪个任务是活跃的，哪些任务是睡眠的。特别要指出的是，位于 0x18010 的 L1 主页表项在上下文切换时是要改变的，使它指向下一个活跃任务的 L2 页表的基地址。

14.10.5 第 5 步：定义页表和区域数据结构

例子中定义了 2 个数据结构，用来配置和控制系统。这 2 个数据结构描绘了用来定义和初始化页表及区域的实际代码。这里定义 2 个数据类型：Pagetable 类型包含页表数据；Region 类型定义和控制系统的每个区域。

Pagetable 结构的类型定义以及结构成员的描述如下：

```
typedef struct {
    unsigned int vAddress;
    unsigned int ptAddress;
    unsigned int masterPtAddress;
    unsigned int type;
    unsigned int dom;
} Pagetable;
```

- vAddress 表示由段项或 L2 页表控制的虚存 1 MB 段的起始地址。
- ptAddress 是页表在虚存中的地址。
- masterPtAddress 是父亲 L1 主页表的地址，如果页表为 L1 页表，则该值与 ptAddress 的值相同。
- type 表示页表的类型，可以是 COARSE(粗)、FINE(细)或 MASTER(主)。
- dom 用来设置分配给 L1 页表项的 1 MB 存储块的域。

这里使用 Pagetable 类型来定义系统中使用的 5 个页表。这些 Pagetable 结构一起组成一块页表数据，用来管理、填充、定位、识别所有活跃和非活跃页表并为其设置域。在示例的其它部分，称这个 Pagetable 块为页表控制块 *PTCB*(Page Table Control Block)。

前面章节中描述(如图 14.20 所示)的 5 个 Pagetable 及其初始值如下：

```
#define FAULT      0
#define COARSE     1
#define MASTER     2
```



```
#define FINE      3

/* 页表 */
/* VADDRESS, PTADDRESS, MasterPTADDRESS, PTTYPE, DOM */
Pagetable masterPT    = {0x00000000, 0x18000, 0x18000, MASTER, 3};
Pagetable systemPT    = {0x00000000, 0x1c000, 0x18000, COARSE, 3};
Pagetable task1PT     = {0x00400000, 0x1c400, 0x18000, COARSE, 3};
Pagetable task2PT     = {0x00400000, 0x1c800, 0x18000, COARSE, 3};
Pagetable task3PT     = {0x00400000, 0x1cc00, 0x18000, COARSE, 3};
```

Region 结构的类型定义,以及结构成员的描述如下:

```
typedef struct {
    unsigned int vAddress;
    unsigned int pageSize;
    unsigned int numPages;
    unsigned int AP;
    unsigned int CB;
    unsigned int pAddress;
    Pagetable * PT;
} Region;
```

- vAddress 是区域在虚存中的起始地址。
- pageSize 是虚存页的大小。
- numPages 是区域中页的数目。
- AP 是区域的访问权限。
- CB 是区域的 cache 和写缓冲器属性。
- pAddress 是区域在物理存储器中的起始地址。
- * PT 是指向区域所对应(reside)的 Pagetable 的指针。

所有的 Region 数据结构一起组成第二个数据块,用来定义系统中使用的区域的大小、位置、访问权限、cache 和写缓冲器操作以及页表位置。在示例的其它部分,称这个 Region 块为区域控制块 RCB(Region Control Block)。

共有 7 个 Region 结构,用来定义在前面章节中描述(如图 14.19 所示)的区域。下面是 RCB 的 4 个系统软件和 3 个任务 Region 的初始值:

```
#define NANA      0x00
#define RWNA      0x01
#define RWRO      0x02
```

ARM 系嵌入式统开发

```

#define RWRW    0x03
/* NA = 不可访问, RO = 只读, RW = 读/写 */

#ifdef __TARGET_CPU_ARM920T
#define cb      0x0
#define cB      0x1
#define WT      0x2
#define WB      0x3
#endif
/* 720 */
#ifdef __TARGET_CPU_ARM720T
#define cb      0x0
#define cB      0x1
#define Cb      0x2
#define WT      0x3
#endif

/* cb = 不使用 cache/不使用写缓冲器 */
/* cB = 不使用 cache/使用写缓冲器 */
/* Cb = 使用 cache/不使用写缓冲器 */
/* WT = 直写策略 cache */
/* WB = 回写策略 cache */

/* 区域表 */
/* VADDRESS, PAGESIZE, NUMPAGES, AP, CB, PADDRESS, &PT */
Region kernelRegion
    = {0x00000000, 4, 16, RWNA, WT, 0x00000000, &systemPT};
Region sharedRegion
    = {0x00010000, 4, 8, RWRW, WT, 0x00010000, &systemPT};
Region pageTableRegion
    = {0x00018000, 4, 8, RWNA, WT, 0x00018000, &systemPT};
Region peripheralRegion
    = {0x10000000, 1024, 256, RWNA, cb, 0x10000000, &masterPT};

/* 任务区域 */
Region t1Region
    = {0x00400000, 4, 8, RWRW, WT, 0x00020000, &task1PT};
Region t2Region
    = {0x00400000, 4, 8, RWRW, WT, 0x00028000, &task2PT};

```

```
Region t3Region
```

```
= {0x00400000, 4, 8, RWRW, WT, 0x00030000, &task3PT};
```

14.10.6 第6步:初始化 MMU、Cache 和写缓冲器

在激活 MMU、cache 和写缓冲器之前,必须对它们进行初始化。PTCB 和 RCB 保存有这 3 个部件的配置信息。初始化 MMU 需要以下 5 个步骤:

- ① 初始化主存中的页表,将它们都填充成 FAULT 项;
- ② 使用将区域映射到物理存储器的转换数据填充页表;
- ③ 激活页表;
- ④ 分配域的访问权限;
- ⑤ 使能存储管理单元和 cache 硬件。

前 4 个步骤配置系统,最后一个步骤使能系统。下面的章节将提供在初始化过程中完成这 5 个步骤的例程,例程的函数和例子号在图 14.21 中列出。

- ① 初始化主存中的页表,将它们都填充成 FAULT 项

```
MmuInitPT(Pagetable *); 例 14.5
```

- ② 使用将区域映射到物理存储器的转换数据填充页表

```
mmuMapRegion(Region *region); 例 14.6
```

```
mmuMapSectionTableRegion(Region *region); 例 14.7
```

```
mmuMapCoarseTableRegion(Region *region); 例 14.8
```

```
mmuMapFineTableRegion(Region *region); 例 14.9
```

- ③ 激活页表

```
int mmuAttachPT(Pagetable *pt); 例 14.10
```

- ④ 分配域访问权限

```
domainAccessSet(unsigned int value, unsigned int mask); 例 14.11
```

- ⑤ 使能存储管理单元和 cache 硬件。

```
controlSet(unsigned int value, unsigned int mask) 例 14.3
```

图 14.21 MMU 初始化例程列表

14.10.6.1 初始化主存中的页表

初始化 MMU 的第 1 步是,将页表设置到一个已知的状态。最简单的方法是使用 FAULT 页表项填充这些页表,这样就确保除了 PTCB 中定义的那些转换数据外,没有其它有效的转换数据存在。将所有活跃页表的所有页表项设置成 FAULT,这样,如果某个页表项没有使用 PTCB 中的数据填充,那么对它访问将产生一个中止异常。

【例 14.5】 例程 mmuInitPT 初始化页表,通过接受分配给页表的存储空间,将它设置成 FAULT 值。它的函数原型如下:

```
void mmuInitPT(Pagetable * pt);
```

例程接受一个指向 PTCB 中 Pagetable 的指针作为参数。

```
int mmuInitPT(Pagetable * pt)
{
    int index;                                /* 每次循环写 PT/entries 中行的数目 */
    unsigned int PTE, * PTEptr;              /* 指向页表中的页表项 */
    PTEptr = (unsigned int *) pt->ptAddress; /* 指向页表基地址 */
    PTE = FAULT;

    switch (pt->type)
    {
        case COARSE: {index = 256/32; break;}
        case MASTER: {index = 4096/32; break;}
        # if defined(__TARGET_CPU_ARM920T)
        case FINE: {index = 1024/32; break;} /* ARM 720T 中没有细(FINE)页表 */
        # endif
        default:
        {
            printf("mmuInitPT: UNKNOWN pagetable type\n");
            return -1;
        }
    }
    __asm
    {
        mov    r0, PTE
        mov    r1, PTE
        mov    r2, PTE
```

```

    mov    r3, PTE
}
for (; index != 0; index--)
{
    __asm
    {
        STMIA PTEptr!, {r0 - r3}          /* 将 32 个项写入表中 */
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
        STMIA PTEptr!, {r0 - r3}
    }
}
return 0;
}

```

mmulnitPT 从页表基地址 PTEptr 开始,使用 FAULT 项填充页表。页表大小通过读取 pt->type 中定义的 Pagetable 的类型确定。页表类型可以是有 4 096 个项的 L1 主页表,有 256 个项的 L2 粗页表,或是有 1 024 个项的 L2 细页表。

例程通过使用循环,每次写一小块存储器来填充页表。页表的项数除以每次循环写的项数就是块的数目,保存到 index 变量中。switch 语句判断 Pagetable 类型,然后跳转到一个设置页表的 index 变量值的 case。执行填充页表的循环以后,程序就结束了。_asm 关键字用于调用内嵌的(inline)汇编语句,这样可以通过使用 stmia 多寄存器装载指令缩短循环的执行时间。

14.10.6.2 用转换数据填充页表

初始化 MMU 的第 2 步是将保存在 RCB 中的数据转化成页表项,然后将其复制到页表中。这里提供多个例程来将 RCB 中的数据转化成页表中的项。最上层的例程 mmuMapRegion 判断页表的类型,然后调用 3 个例程中的一个来构造页表项:mmuMapSectionTableRegion,mmuMapCoarseTableRegion 或 mmuMapFineTableRegion。

为使以后代码的移植更容易,建议不要使用微(tiny)页和 mmuMapFineTableRegion 例程,因为 ARMv6 体系结构不再使用微(tiny)页。细(fine)页表类型也已从 ARMv6 体系结构中删除了,因为如果没有微页,也就不需要细(fine)页表了。

下面是 4 个例程的描述:

- mmuMapRegion 例程判断页表的类型,然后分支到下面列出的 3 个例程之一:在例 14.6 介绍。
- mmuMapSectionTableRegion 使用段项填充 L1 主页表:在例 14.7 介绍。
- mmuMapCoarseTableRegion 使用区域项填充 L2 粗页表:在例 14.8 介绍。
- mmuMapFineTableRegion 使用区域项填充 L2 细页表:在例 14.9 介绍。

下面是 4 个例程的 C 函数原型列表:

```
int mmuMapRegion(Region *);  
void mmuMapSectionTableRegion(Region * region);  
int mmuMapCoarseTableRegion(Region * region);  
int mmuMapFineTableRegion(Region * region);
```

4 个例程都有一个参数,指向包含生成页表项所需的配置数据的 Region 结构指针。

【例 14.6】 是最上层的例程,判断页表类型。

```
int mmuMapRegion(Region * region)  
{  
    switch (region->PT->type)  
    {  
        case SECTION;                                /* L1 页表段项 */  
        {  
            mmuMapSectionTableRegion(region);  
            break;  
        }  
        case COARSE;                                  /* L2 粗页表 */  
        {  
            mmuMapCoarseTableRegion(region);  
            break;  
        }  
        # if defined(__TARGET_CPU_ARM920T)  
        case FINE;                                    /* L2 细页表 */  
        {  
            mmuMapFineTableRegion(region);  
            break;  
        }  
        #endif  
        default;  
        {
```

```

    printf("mmuMapRegion: UNKNOWN pagetable type\n");
    return -1;
}
}
return 0;
}

```

Region 中有一个指向 Pagetable 的指针,区域的转换数据就保存在这里。例程判断页表类型 `region->PT->type`,然后调用一个例程,以给定页表类型的格式将 Region 映射到页表中。

对于段(L1 主)、粗和细 3 种页表分别有单独的例程(参见 14.4 节)。

【例 14.7】 是第 1 个例程,把区域数据转化成页表项。

```

void mmuMapSectionTableRegion(Region * region)
{
    int i;
    unsigned int * PTEptr, PTE;

    PTEptr = (unsigned int *)region->PT->ptAddress; /* 页表起始地址 */
    PTEptr += region->vAddress >> 20; /* 区域的第一个页表项 */
    PTEptr += region->numPages - 1; /* 区域的最后一个页表项 */

    PTE = region->pAddress & 0xfff00000; /* 设置起始物理地址 */
    PTE |= (region->AP & 0x3) << 10; /* 设置访问权限 */
    PTE |= region->PT->dom << 5; /* 设置段的域 */
    PTE |= (region->CB & 0x3) << 2; /* 设置 cache 和写缓冲器属性 */
    PTE |= 0x12; /* 段项 */

    for (i = region->numPages - 1; i >= 0; i--) /* 填充区域的页表项 */
    {
        *PTEptr-- = PTE + (i << 20); /* i=1 MB 段 */
    }
}

```

在 `mmuMapSectionTableRegion` 例程的开始,先设置局部指针变量 `PTEptr`,使它指向 L1 主页表的基地址。然后例程使用区域的虚拟起始地址来创建一个页表的索引,区域的页表项从索引处开始。这个索引值加到 `PTEptr` 变量中,这时变量 `PTEptr` 就指向页表中区域项的起始位置。下一个程序行计算区域的大小,然后将这个值加到 `PTEptr` 中。这时

PTEptr 指向区域的最后一个页表项(PTE)。PTEptr 变量被设置为区域的结束位置,因此在填充页表的循环中可以使用向下递减(count-down)计数器。

接下来,例程使用 Region 结构中的值构造一个段页表项,这个项保存在局部变量 PTE 中。一系列 OR 语句设置这个 PTE,包括起始物理地址、访问权限、域以及 cache 和写缓冲器属性。PTE 的格式如图 14.6 所示。

这时 PTE 包含指向区域的起始物理地址和属性的指针。计数器变量 i 有 2 个作用:首先它是页表偏移量(offset);其次它加到 PTE 变量中,以递增页帧的物理地址转换数据。这里须指出的是,示例中所有的区域都映射到物理存储器连续的页帧。从最后一个转换项开始,然后向下计数到第一个转换项,程序将区域的所有 PTE 写入页表后结束。

【例 14.8】 2 个例程:mmuMapCoarseTableRegion 和 mmuMapFineTableRegion,非常相似,使得例程的文字描述也几乎相同。看了粗页表的例子后,如果不使用微页,可以跳过细页表的例子。

```
int mmuMapCoarseTableRegion(Region * region)
{
    int i,j;
    unsigned int * PTEptr, PTE;
    unsigned int tempAP = region->AP & 0x3;

    PTEptr = (unsigned int *)region->PT->ptAddress;    /* PTEptr = 页表起始地址 */

    switch (region->pageSize)
    {
        case LARGEPAGE;
        {
            PTEptr += (region->vAddress & 0x000ff000) >> 12; /* 第一个页表项 */
            PTEptr += (region->numPages * 16) - 1;           /* 区域最后一个页表项 */

            PTE = region->pAddress & 0xffff0000;             /* 设置物理地址 */
            PTE |= tempAP << 10;                             /* 设置访问权限 subpage 3 */
            PTE |= tempAP << 8;                               /* subpage 2 */
            PTE |= tempAP << 6;                               /* subpage 1 */
            PTE |= tempAP << 4;                               /* subpage 0 */
            PTE |= (region->CB & 0x3) << 2;                  /* 设置 cache 和写缓冲器属性 */
            PTE |= 0x1;                                       /* set as LARGE PAGE */
        }
    }
}
```



```

/* 填充区域页表项 */
for (i = region->numPages - 1; i >= 0; i--)
{
    for (j = 15; j >= 0; j--)
        *PTEptr-- = PTE + (i << 16); /* i = 64 KB large page */
    break;
}
case SMALLPAGE;
{
    PTEptr += (region->vAddress & 0x000ff000) >> 12; /* 第一个页表项 */
    PTEptr += (region->numPages - 1); /* 最后一个页表项 */

    PTE = region->pAddress & 0xfffff000; /* 设置物理地址 */
    PTE |= tempAP << 10; /* 设置访问权限 subpage 3 */
    PTE |= tempAP << 8; /* subpage 2 */
    PTE |= tempAP << 6; /* subpage 1 */
    PTE |= tempAP << 4; /* subpage 0 */
    PTE |= (region->CB & 0x3) << 2; /* 设置 cache 和写缓冲器属性 */
    PTE |= 0x2; /* set as SMALL PAGE */

    /* 填充区域页表项 */

    for (i = region->numPages - 1; i >= 0; i--)
    {
        *PTEptr-- = PTE + (i << 12); /* i = 4 KB small page */
    }
    break;
}
default;
{
    printf("mmuMapCoarseTableRegion: Incorrect page size\n");
    return -1;
}
}
return 0;
}

```

在例程的开始,先设置局部变量 tempAP,使它保存区域中页或子页的访问权限。接着

设置变量 PTEptr, 使它指向将用来保存映射区域页表的基地址。

然后例程分别选择处理大 (large) 或小 (small) 页的情况, 两种情况所使用的算法是一样的, 只是 PTE 的格式和写入页表的条目不同。

这时变量 PTEptr 包含 L2 页表的起始地址。然后例程使用区域的起始地址 region->vAddress 计算页表中区域的第一个项的索引, 将这个索引值加到 PTEptr 中。下一个程序行计算区域的大小, 将该值加到 PTEptr 中。这时 PTEptr 指向区域的最后一个页表项 (PTE)。

接下来以传给例程的区域中的值, 为大或小的页表项构造页表项变量 PTE。例程使用一系列的 OR 语句来构造 PTE, 包括起始物理地址、访问权限以及 cache 和写缓冲器属性。可以参见图 14.8 复习一下大和小的 PTE 格式。

这时 PTE 包含指向区域的第一个页帧的物理地址的指针。计数器变量 i 有 2 个作用: 首先它是页表偏移量 (offset); 其次它加到 PTE 变量中, 以修改地址转换位域, 使 PTE 指向物理存储器中下一个低地址的页帧。例程将区域的所有 PTE 写入页表后结束。特别要指出的是, LARGE PAGE 这个 case 中有一个嵌套的循环——j 循环, 它通过写入所需相同的 PTE 来映射粗页表中大的页 (详细内容请参见 14.4 节)。

【例 14.9】 使用区域转换信息填充细页表。

ARM720T 中没有细页表, ARM v6 体系结构也不再使用细页表。为了与这些改变相兼容, 建议在新项目中不要再使用细页表。

```
# if defined(__TARGET_CPU_ARM920T)
int mmuMapFineTableRegion(Region * region)
{
    int i, j;
    unsigned int * PTEptr, PTE;
    unsigned int tempAP = region->AP & 0x3;

    PTEptr = (unsigned int *)region->PT->ptAddress;    /* 页表起始地址 */

    switch (region->pageSize)
    {
        case LARGE PAGE:
        {
            PTEptr += (region->vAddress & 0x000ffc00) >> 10; /* 第一个页表项 */
            PTEptr += (region->numPages * 64) - 1;           /* 最后一个页表项 */

            PTE = region->pAddress & 0xffff0000;             /* 物理地址 */
        }
    }
}
```

```

PTE |= tempAP << 10; /* 设置访问权限 subpage 3 */
PTE |= tempAP << 8; /* subpage 2 */
PTE |= tempAP << 6; /* subpage 1 */
PTE |= tempAP << 4; /* subpage 0 */
PTE |= (region->CB & 0x3) << 2; /* 设置 cache 和写缓冲器属性 */
PTE |= 0x1; /* set as LARGE PAGE */

/* 填充区域页表项 */

for (i = region->numPages - 1; i >= 0; i--)
{
    for (j = 63; j >= 0; j--)
        *PTEptr-- = PTE + (i << 16); /* i = 64 KB large page */
    break;
}

case SMALLPAGE;
{
    PTEptr += (region->vAddress & 0x000ffc00) >> 10; /* 第一个页表项 */
    PTEptr += (region->numPages * 4) - 1; /* 最后一个页表项 */

    PTE = region->pAddress & 0xfffff000; /* 物理地址 */
    PTE |= tempAP << 10; /* 设置访问权限 subpage 3 */
    PTE |= tempAP << 8; /* subpage 2 */
    PTE |= tempAP << 6; /* subpage 1 */
    PTE |= tempAP << 4; /* subpage 0 */
    PTE |= (region->CB & 0x3) << 2; /* 设置 cache 和写缓冲器属性 */
    PTE |= 0x2; /* set as SMALL PAGE */

    /* 填充区域页表项 */

    for (i = region->numPages - 1; i >= 0; i--)
    {
        for (j = 3; j >= 0; j--)
            *PTEptr-- = PTE + (i << 12); /* i = 4 KB small page */
        break;
    }
}

case TINYPAGE;

```

```

{
    PTEptr += (region->vAddress & 0x000ffc00) >> 10; /* 第一个页表项 */
    PTEptr += (region->numPages - 1); /* 最后一个页表项 */

    PTE = region->pAddress & 0xfffffc00; /* 物理地址 */
    PTE |= tempAP << 4; /* 设置访问权限 */
    PTE |= (region->CB & 0x3) << 2; /* 设置 cache 和写缓冲器属性 */
    PTE |= 0x3; /* set as TINY PAGE */

    /* 从最后一个页表项到第一个页表项,填充区域页表 */
    for (i = (region->numPages) - 1; i >= 0; i--)
    {
        *PTEptr++ = PTE + (i << 10); /* i = 1 KB tiny page */
    }
    break;
}
default:
{
    printf("mmuMapFineTableRegion: Incorrect page size\n");
    return -1;
}
}

return 0;
}

#endif

```

在例程的开始,先设置局部变量 tempAP,使它保存区域中页或子页的访问权限。这个例程不支持子页拥有不同的访问权限。接着设置变量 PTEptr,使它指向页表的基地址,这个页表将用来保存映射到细分页的区域。

然后,例程选择处理大、小或微页的 3 种情况。3 个 case 所使用的算法是一样的,只是 PTE 的格式和写入页表条目不同。

这时变量 PTEptr 包含 L2 页表的起始地址。然后例程使用区域的起始地址 region->vAddress 计算页表中第一个区域项的索引,将这个索引值加到 PTEptr 中。下一个程序行确定区域的大小,将该值加到 PTEptr 中。现在 PTEptr 指向区域的最后一个页表项 (PTE)。

接下来例程以区域中的值,为大、小或微页表项构造 PTE。例程使用一系列 OR 语句来构造一个 PTE,包括起始物理地址、访问权限以及 cache 和写缓冲器属性。图 14.8 描述

了大、小和微页表项的格式。

这时 PTE 包含了指向区域的第一个页帧的物理地址和属性的指针。计数器变量 i 有 2 个作用:首先它是页表偏移量(offset);其次它加到 PTE 变量中,以修改地址转换位域,使得 PTE 指向物理存储器中下一个低地址的页帧。执行循环,直到将区域的所有 PTE 写入页表中,然后例程结束。特别要指出的是,LARGE PAGE 和 SMALL PAGE 这 2 个 case 中有一个嵌套的循环—— j 循环,它写入所需相同的 PTE 来映射细页表中的给定页。

14.10.6.3 激活页表

页表可以保存在存储器中,而不被 MMU 硬件使用。这种情况发生在当任务处于睡眠态时,其页表被映射到活跃的虚存之外。然而任务仍驻留在物理存储器中,因此当发生上下文切换激活这个任务时,它立即又可以使用。

初始化 MMU 的第 3 步是,为了执行位于特定区域的代码,激活所需的页表。

【例 14.10】 例程 mmuAttachPT 或者激活一个 L1 主页表(通过将其地址放到 CP15: c2:c0 寄存器的 TTB 中);或者激活一个 L2 页表(通过将其基地址放到 L1 主页表项中)。

它的函数原型如下:

```
int mmuAttachPT(Pagetable * pt);
```

例程只接受一个参数——指向待激活 Pagetable 的指针,并增加新的从虚存到物理存储器的转换数据。

```
int mmuAttachPT(Pagetable * pt)                /* 把 L2 PT 附于 L1 主 PT */
{
    unsigned int * ttb, PTE, offset;

    ttb = (unsigned int *)pt->masterPtAddress; /* 从 PT 中读 ttb */
    offset = (pt->vAddress) >> 20;             /* 从 vAddress 决定 PTE */

    switch (pt->type)
    {
        case MASTER:
        {
            __asm{ MCR p15, 0, ttb, c2, c0, 0 }; /* TTB -> CP15:c2:c0 */
            break;
        }
        case COARSE:
        {
            /* PTE = addr L2 PT | domain | COARSE PT type */

```

ARM 系嵌入式统开发

```

        PTE = (pt->ptAddress & 0xfffffc00);
        PTE |= pt->dom << 5;
        PTE |= 0x11;
        ttb[offset] = PTE;
        break;
    }
    # if defined(__TARGET_CPU_ARM920T)
    case FINE:
    {
        /* PTE = addr L2 PT | domain | FINE PT type */
        PTE = (pt->ptAddress & 0xffff000);
        PTE |= pt->dom << 5;
        PTE |= 0x13;
        ttb[offset] = PTE;
        break;
    }
    # endif
    default:
    {
        printf("mmuAttachPT: UNKNOWN pagetable type\n");
        return -1;
    }
}

return 0;
}

```

该例程做的第一件事是,准备 2 个变量:L1 主页表的基地址 ttb 和 L1 页表偏移量 offset。变量 offset 由页表的虚地址产生。为了计算偏移量,例程读取虚地址,然后除以 1 MB,这通过将虚地址右移 20 位来实现。将这个偏移量加到 L1 主页表的基地址,产生一个指针,指向 L1 主页表中代表 1 MB 段的转换数据的地址。

例程用 Pagetable 类型的 pt->type 变量来跳转到相关页表的 case,将页表系于(attach)MMU 硬件。3 个可能的 case 将在下面描述。

- MASTER 处理 L1 主页表。例程使用汇编语言的 MCR 指令来设置 CP15:c2:c0 寄存器,处理这个特殊的页表。
- COARSE 把一个粗页表系于 L1 主页表中。这个 case 读取保存在 Pagetable 结构中的 L2 页表地址,组合结构的 dom 成员和粗页表类型,构造 PTE。然后将 PTE 写入

L1 页表,这使用前面计算得到的偏移值。粗 PTE 的格式如图 14.6 所示。

- *FINE* 把一个 L2 细页表系于 L1 主页表中。这个 case 读取保存在 Pagetable 结构中的 L2 页表地址,组合结构的 dom 成员和细页表类型,构造 PTE。然后使用前面计算得到的偏移值,将 PTE 写入 L1 页表。

以上章节给出了初始化 MMU 时判断、装载和激活页表的例程。下面将设置域访问权限并使能 MMU。

14.10.6.4 分配域访问权限和使能 MMU

初始化 MMU 的第 4 步是配置系统的域(domain)访问权限。示例中不使用 FCSE,也不需要快速显现(expose)和隐藏(hide)大块的存储器,因此不需要使用 CP15:c1:c0 寄存器中的 S 和 R 访问控制位。这意味着页表中定义的访问权限对保护系统来说已经足够,并且应该使用域。

然而,硬件需要所有活跃的存储空间有域的分配,并赋予域访问权限。最简单的域配置是将所有的区域(region)放在同一个域(domain)里,把域的访问权限设置为客户访问允许。在域的这种配置下,只有页表中允许访问的项才是系统活跃项。

示例中,所有的区域都分配在域 3,并分配客户访问允许权限。其它的域不使用,并通过 L1 主页表的未使用页表项中的错误(fault)项,将不使用的域屏蔽掉。域在 L1 主页表中分配,域访问权限在 CP15:c3:c0 寄存器中定义。

【例 14.11】 domainAccessSet 例程设置域访问控制寄存器 CP15:c3:c0:0 中 16 个域的访问权限。它的函数原型如下:

```
void domainAccessSet(unsigned int value, unsigned int mask);
```

传递给例程的第 1 个参数是一个无符号整数,包含设置 16 个域的域访问权限的位域。第 2 个参数决定需要改变哪些域的访问权限。例程首先读 CP15:c3 寄存器,将其值保存到 c3format 变量中;然后使用输入的 mask 值清除 c3format 中需要更新的位,这是通过把变量 c3format 和输入的 value 值做 OR 运算来完成的;最后将更新后的 c3format 值回写到 CP15:c3 寄存器,完成设置域访问权限。

```
void domainAccessSet(unsigned int value, unsigned int mask)
{
    unsigned int c3format;

    __asm{MRC p15, 0, c3format, c3, c0, 0} /* 读域寄存器 */
    c3format &= ~mask;                      /* 清除要改变的位 */
    c3format |= value;                     /* 设置要改变的位 */
}
```

```
__asm{MCR p15, 0, c3format, c3, c0, 0} /* 写域寄存器 */
```

使能 MMU 是初始化 MMU 过程的第 5 步,也是最后一步。例程 controlSet(例 14.3)使能 MMU。建议在一个“固定”的地址空间中调用 controlSet 例程。

14.10.6.5 完成初始化 MMU

例程 mmuInit 调用前面章节中描述的例程,初始化示例的 MMU。读这段代码时,可以先复习一下 14.10.5 小节介绍的控制块。

例程的 C 函数原型如下:

```
void mmuInit(void);
```

【例 14.12】 调用前面描述的初始化 MMU 的 5 个步骤。5 个步骤在例子代码中以注释的形式标出。

在例程 mmuInit 的开始,先初始化页表,并将区域映射到特权的系统区域。第 1 步是调用 mmuInitPT 例程,初始化固定的系统空间。这些调用使用 FAULT 值填充 L1 主页表和 L2 页表。例程调用 mmuInitPT 5 次:一次初始化 L1 主页表;一次初始化系统 L2 页表;然后 3 次调用 mmuInitPT 初始化 3 个任务页表:

```
#define DOM3CLT          0x00000040
#define CHANGEALLDOM     0xffffffff

#define ENABLEMMU        0x00000001
#define ENABLEDCACHE     0x00000004
#define ENABLEICACHE     0x00001000
#define CHANGEMMU        0x00000001
#define CHANGEDCACHE     0x00000004
#define CHANGEICACHE     0x00001000
#define ENABLEWB         0x00000008
#define CHANGEWB         0x00000008
```

```
void mmuInit()
```

```
{
```

```
    unsigned int enable, change;
```

```
    /* 第 1 步 初始化系统(固定的)页表 */
```

```
    mmuInitPT(&masterPT);          /* 初始化主 L1 PT 和 FAULT PTE */
```

```
    mmuInitPT(&systemPT);          /* 初始化系统 L2 PT 和 FAULT PTE */
```



```

mmuInitPT(&task3PT);          /* 初始化任务 3 L2 PT 和 FAULT PTE */
mmuInitPT(&task2PT);          /* 初始化任务 2 L2 PT 和 FAULT PTE */
mmuInitPT(&task1PT);          /* 初始化任务 1 L2 PT 和 FAULT PTE */

/* 第 2 步 使用转换和属性数据填充页表 */
mmuMapRegion(&kernelRegion);   /* 映射 kernelRegion SystemPT */
mmuMapRegion(&sharedRegion);    /* 映射 sharedRegion SystemPT */
mmuMapRegion(&pageTableRegion); /* 映射 pagetableRegion SystemPT */
mmuMapRegion(&peripheralRegion); /* 映射 peripheralRegion MasterPT */

mmuMapRegion(&t3Region);        /* 映射任务 3 PT 和区域数据 */
mmuMapRegion(&t2Region);        /* 映射任务 2 PT 和区域数据 */
mmuMapRegion(&t1Region);        /* 映射任务 2 PT 和区域数据 */

/* 第 3 步 激活页表 */
mmuAttachPT(&masterPT);        /* 将 L1 TTB 装载到 CP15:c2:c0 寄存器 */
mmuAttachPT(&systemPT);        /* 将 L2 系统 PTE 装载到 L1 PT */
mmuAttachPT(&task1PT);         /* 将 L2 任务 1 PTE 装载到 L1 PT */

/* 第 4 步 设置域访问 */

domainAccessSet(DOM3CLT, CHANGEALLDOM); /* 设置域访问 */

/* 第 5 步 使能 MMU、caches 和写缓冲器 */
#ifdef __TARGET_CPU_ARM720T
    enable = ENABLEMMU | ENABLECACHE | ENABLEWB;
    change = CHANGEMMU | CHANGECACHE | CHANGEWB;
#endif
#ifdef __TARGET_CPU_ARM920T
    enable = ENABLEMMU | ENABLEICACHE | ENABLEDCACHE;
    change = CHANGEMMU | CHANGEICACHE | CHANGEDCACHE;
#endif
controlSet(enable, change);      /* 使能 cache 和 MMU */
}

```

第 2 步通过调用 mmuMapRegion 例程 7 次, 将系统的 7 个区域映射到其页表中: 4 次映射内核、共享、页表和外设区域, 3 次映射 3 个任务区域。mmuMapRegion 将控制块中的数据转化成页表项, 然后将这些页表项写入页表。

初始化 MMU 的第 3 步是激活启动系统所需的页表,这是通过 3 次调用 mmuAttachPT 例程来完成的。例程首先将 L1 主页表的基地址装载到 CP15:c2:c0 的 TTB 来激活 L1 主页表,然后激活 L2 系统页表。外设区域由保存在 L1 主页表中的 1 MB 页组成,在 L1 主页表激活时它就被激活了。最后调用 mmuAttachPT 例程,激活系统启动后将要运行的第一个任务。这个示例中,要运行的第一个任务是任务 1。

初始化 MMU 的第 4 步是调用 domainAccessSet 例程,设置域访问权限。所有的区域都分配到域 3,域 3 的域访问权限设置成客户访问允许。

例程 mmuInit 最后调用 controlSet 例程,使能 MMU 和 cache。

例程 mmuInit 执行完后,MMU 初始化完毕并被使能了。建立多任务示例系统的最后一个任务是,定义在 2 个任务间执行上下文切换所需要的程序。

14.10.7 第 7 步:建立上下文切换程序

示例系统的上下文切换相对来说比较简单,主要有以下 6 个步骤:

- ① 保存活跃任务的上下文,并将活跃任务置为睡眠态;
- ② 清除 cache,如果使用回写策略,则还要清理数据 cache;
- ③ 清除 TLB,以删除退出任务的转换数据;
- ④ 配置 MMU,以使用新的页表,把同样的虚存执行空间转换到唤醒任务在物理存储器中的位置;
- ⑤ 恢复唤醒任务的上下文;
- ⑥ 开始执行恢复的任务。

执行以上步骤的例程已在前面章节中介绍过了,这里只是简单地列出这些例程。第 1, 5 和 6 步在第 11 章介绍过;详细内容可以参见第 11 章。第 2,3 和 4 步是使用 MMU 上下文切换需要增加的部分,与其一起列出的是示例中从任务 1 切换到任务 2 所需的参数。

```
SAVE retiring task context;           /* 第 1 步在第 11 章中介绍 */
flushCache();                        /* 第 2 步在第 12 章中介绍 */
flushTLB();                          /* 第 3 步在例 14.2 中介绍 */
mmuAttachPT(&task2PT);              /* 第 4 步在例 14.10 中介绍 */
RESTORE awakening task context       /* 第 5 步在第 11 章中介绍 */
RESUME execution of restored task   /* 第 6 步在第 11 章中介绍 */
```

14.11 MMUSLOS 示例

MMU 示例代码中的许多概念和例子被包含到一个实用的控制系统中,该系统被称为 mmuSLOS。它是第 11 章介绍的 SLOS 控制系统的扩展。

mpuSLOS 是 SLOS 带存储保护单元的扩展,已经在第 13 章中介绍过了。这里使用 mpuSLOS 作为 mmuSLOS 的基本源代码。与 mpuSLOS 相比,mmuSLOS 主要有以下 3 个变化。

- MMU 页表是在 mmuSLOS 的初始化阶段建立的。
- 应用程序任务被构建成在地址 0x400000 处执行,但是它们被装载在不同的物理地址上。每个应用程序任务从它的执行地址开始,在一个虚存中执行。栈顶被定位在执行地址的 32 KB 偏移处。
- 每次调用调度器,都改变 MMU 页表中活跃的 32 KB 页,以反映出新的活跃的应用或任务。

14.12 总 结

本章介绍了存储器管理和虚拟存储器系统的基本知识。MMU 的一个主要服务是能把各个任务作为各自独立的程序在其自己的虚拟存储空间中运行。

虚拟存储器系统的一个重要特征是地址重定位。地址重定位是将处理器核产生的地址转换到主存的不同地址,转换由 MMU 硬件完成。

在一个虚拟存储器系统中,虚存通常作为固定空间或动态空间被分成几个部分。在固定空间内,映射在页表中的转换数据在普通操作中不发生变化;在动态空间内,虚存到物理存储器之间的映射关系频繁发生变化。

页表包含虚拟页的描述信息。一个页表项(PTE)将虚存中的一页转换成物理存储器中的一个页帧。页表项通过虚地址进行组织,包含将一页转换成一个页帧的转换数据。

ARM MMU 的功能如下:

- 读 L1 和 L2 页表,并将其装载到 TLB 中;
- 在 TLB 中保存最近的虚—实地址转换数据;
- 执行虚地址到实地址的转换;
- 强化访问权限,配置 cache 和写缓冲器。

ARM MMU 中增加的一个特殊功能是快速上下文切换扩展(FCSE)。快速上下文切换扩展改善了多任务环境中的系统性能,因为在上下文切换时,它不需要清除 cache 和 TLB。

一个简单的虚拟存储器系统的实用例子详细介绍了配置 MMU,以支持多任务的过程。配置步骤为:定义虚存的固定系统软件所使用的区域;定义每个任务的虚拟存储映射;将任务的区域定位到物理存储器映射中;在页表区域中定义和定位页表;定义创建、管理区域和页表所需要的数据结构;初始化 MMU;使用预定义的区域数据构造页表项,并将它们写入页表中;建立上下文切换例程,从一个任务转换到另一个任务。

第 15 章

ARM 体系结构的发展

- ARM v6 对高级 DSP 和 SIMD 的支持
- ARM v6 增加的系统和多处理器支持
- ARM v6 的实现
- ARM v6 之后的未来技术
- 总 结

ARM 嵌入式系统开发

1999 年 10 月,ARM 开始考虑体系结构的未来发展方向,最终诞生了 ARMv6,并在一个新的产品 ARM1136J-S 中第一次得到了运用。到目前为止,针对许多不同类型的应用,ARM 已经有多种设计。我们不仅需要分析、评估这些应用将来的每一个新要求,同时还须考虑未来 ARM 可能的新的应用领域。

随着片上系统设计变得更加精密、复杂,ARM 处理器已成为包含多个处理部件和子系统的系统核心处理器。特别是便携式和移动计算市场,给 ARM 带来了新的软件和性能的挑战。需要面向的领域包括:针对便携式设备的数字信号处理(DSP)和视频性能;大端-小端混合系统的交互,例如 TCP/IP;多处理器环境下的高效同步等。ARM 面临的挑战是,满足所有这些市场的需求,同时在计算效率(每 mW 的计算能力)方面,继续保持工业界最强的竞争优势。

本章将介绍 ARM 公司针对以上市场需求,在 ARMv6 中引入的新技术和结构组成,包括增强的 DSP 支持和对多处理器环境的支持。另外,还介绍了这种高性能的 ARMv6 的第一个实现,还有 ARM 的最新技术之一——TrustZone。

15.1 ARMv6 对高级 DSP 和 SIMD 的支持

在 ARMv6 项目的早期,ARM 就开始考虑如何在 ARMv5E(3.7 节中描述)扩展的基础上,进一步提高体系结构的 DSP 和媒体处理能力。这项工作的开展,是和早期开发 ARM1136J-S 产品的微体系结构的工程组紧密结合的。单指令流多数据流(SIMD)是一种流行的技术,用于并行地存储大量的数据。对于 DSP 中普遍使用的含有大量复杂数学运算的程序,如视频和图像处理算法,这种技术特别有效。SIMD 最大的优点就是高代码密度和低功耗,因为执行的指令数量很少(同样减少了存储系统的访问)。但这种效率的代价是,降低了灵活性,因为必须为计算安排好固定的块数据格式。不过,这种技术在许多图像和信号处理算法中运用得非常好。

遵循标准的 ARM 低功耗、高计算效率设计原则,ARM 提出了一种简单而又先进的方法,将现存的 32 位 ARM 数据通道划分成 4 个 8 位或 2 个 16 位的片段(slice)。不同于许多现有的 SIMD 结构,为 SIMD 操作增加独立的数据总线;这种方法以最小的硬件代价,把 SIMD 加入到基本的 ARM 体系结构中。

ARMv6 体系结构中包含了这种“轻量级(lightweight)”的 SIMD 方法,在额外的复杂度(门计数)和功耗方面,也几乎无须付出什么代价。同时,新的指令能够提高一些算法的处理吞吐量,对于 16 位的数据最大可达 2 倍;对于 8 位的数据最大可达 4 倍。与 ARM 指令集结构中的大部分操作一样,所有的这些新指令都是条件执行的,如 2.2.6 小节中的描述。

可以在附录 A 的指令集表中找到所有 ARMv6 指令的详细描述。

15.1.1 SIMD 算法操作

表 15.1 中列出了 8 位的 SIMD 操作。源操作数中包含多个以字节为单位的片段,相应的字节片段经过算法操作后,产生一个字节结果。

表 15.1 8 位 SIMD 算法操作

指 令	描 述
SADD8 {<cond>} Rd, Rn, Rm	有符号 8 位 SIMD 加法
SSUB8 {<cond>} Rd, Rn, Rm	有符号 8 位 SIMD 减法
UADD8 {<cond>} Rd, Rn, Rm	无符号 8 位 SIMD 加法
USUB8 {<cond>} Rd, Rn, Rm	无符号 8 位 SIMD 减法
QADD8 {<cond>} Rd, Rn, Rm	有符号饱和 8 位 SIMD 加法
QSUB8 {<cond>} Rd, Rn, Rm	有符号饱和 8 位 SIMD 减法
UQADD8 {<cond>} Rd, Rn, Rm	无符号饱和 8 位 SIMD 加法
UQSUB8 {<cond>} Rd, Rn, Rm	无符号饱和 8 位 SIMD 减法

这些 8 位数据的运算结果,可能最多需要 9 位数据来表示。这样可能会产生回卷 (wraparound)或饱和,取决于具体所使用的特定指令。

除了 8 位的 SIMD 操作之外,还有扩展的双 16 位的操作,如表 15.2 中所列出的。源操作数中包含 2 个 16 位的数据片段,相应的半字片段经过算法操作后,产生一个半字(16 位)结果。

表 15.2 16 位 SIMD 算法操作

指 令	描 述
SADD16 {<cond>} Rd, Rn, Rm	有符号 16 位 SIMD 加法
SSUB16 {<cond>} Rd, Rn, Rm	有符号 16 位 SIMD 减法
UADD16 {<cond>} Rd, Rn, Rm	无符号 16 位 SIMD 加法
USUB16 {<cond>} Rd, Rn, Rm	无符号 16 位 SIMD 减法
QADD16 {<cond>} Rd, Rn, Rm	有符号饱和 16 位 SIMD 加法
QSUB16 {<cond>} Rd, Rn, Rm	有符号饱和 16 位 SIMD 减法
UQADD16 {<cond>} Rd, Rn, Rm	无符号饱和 16 位 SIMD 加法
UQSUB16 {<cond>} Rd, Rn, Rm	无符号饱和 16 位 SIMD 减法

这些 16 位数据的运算结果,可能需要 17 位数据来存储。这样就可能在 16 位有符号数范围内产生回卷或饱和,取决于具体所使用的特定指令。

以上这些 SIMD 指令的操作数,并不总是按照我们希望的顺序存放在源寄存器中。为了提高处理这种情况的效率,另外还有 16 位的 SIMD 操作,用于交换一个操作数寄存器中的高低 16 位。这些操作在处理以不同边界方式存放在存储器中的半字时,非常灵活,并在处理打包存储在 32 位寄存器中的 16 位复数对(complex number pairs)时,也特别有用。这些操作包括有符号数、无符号数、饱和有符号数以及饱和无符号数等形式,如表 15.3 中所列。

表 15.3 16 位带交换 SIMD 算法操作

指令	描述
SADDSUBX {<cond>} Rd, Rn, Rm	有符号高位加法,低位减法,Rm 中半字交换
UADDSUBX {<cond>} Rd, Rn, Rm	无符号高位加法,低位减法,Rm 中半字交换
QADDSUBX {<cond>} Rd, Rn, Rm	有符号饱和高位加法,低位减法,Rm 中半字交换
UQADDSUBX {<cond>} Rd, Rn, Rm	无符号饱和高位加法,低位减法,Rm 中半字交换
SSUBADDX {<cond>} Rd, Rn, Rm	有符号高位减法,低位加法,Rm 中半字交换
USUBADDX {<cond>} Rd, Rn, Rm	无符号高位减法,低位加法,Rm 中半字交换
QSSUBADDX {<cond>} Rd, Rn, Rm	有符号饱和高位减法,低位加法,Rm 中半字交换
UQSUBADDX {<cond>} Rd, Rn, Rm	无符号饱和高位减法,低位加法,Rm 中半字交换

这些指令中的符号 X 意味着 Rm 中的 2 个半字。在进行运算操作之前,应先进行交换操作,如下所示:

$$Rd[15:0] = Rn[15:0] - Rm[31:16]$$

$$Rd[31:16] = Rn[31:16] + Rm[15:0]$$

这些新增的 SIMD 运算意味着需要有一种方法来表示数据通道上每一个 SIMD 片段运算产生的进位或者溢出。在 2.2.5 小节中所描述的 cpsr 寄存器中,增加了 4 个附加的标志,以分别表示数据通道每 8 位数据片段的状态。新的修改后的 cpsr 寄存器见图 15.1 和表 15.4,其中包含了 GE 位。每个 GE 位分别作为数据通道上各个数据片段的“大于或等于”标志。

操作系统已经在上下文切换时保存了 cpsr 寄存器,因此在 cpsr 中加上这些位,对于体系结构对操作系统的支持不会有什么影响。

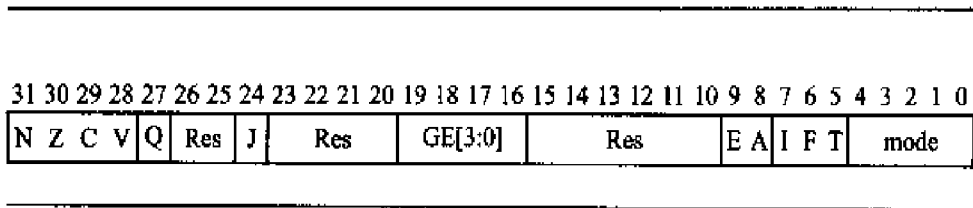


图 15.1 ARMv6 的 cpsr 寄存器

表 15.4 ARMv6 的 cpsr 域

域	使 用
N	负标志。记录标志设置操作结果的 31 位
Z	零标志。当标志设置操作结果为零时置位
C	进位标志。无符号的加法操作的进位、减法操作的借位,同时也用于循环移位操作。见附录表 A.3
V	溢出标志。记录标志设置操作的带符号的溢出
Q	饱和标志。一些饱和操作在产生饱和时置位该标志,详见附录 A 中的 QADD 的例子(ARMv5E 和以上的版本)
J	J=1 表示执行 Java(必须满足 T=0)。使用 BXJ 指令来改变该位的值(ARMv5J 和以上的版本)
Res	这些位保留,用于将来扩展。软件应该保留这些位中的值
GE[3:0]	SIMD 大于或等于标志。详见附录 A 中的 SADD(ARMv6)
E	控制数据的大端和小端。详见附录 A 中的 SETEND(ARMv6)
A	A=1 使不精确数据异常中断无效(ARMv6)
I	I=1 使 IRQ 中断无效
F	F=1 使 FIQ 中断无效
T	T=1 表示 Thumb 状态;T=0 表示 ARM 状态。使用 BX 或 BLX 指令来改变这一位(ARMv4T 和以上的版本)
mode	当前的处理器模式,见附录表 B.4

除了基本的 SIMD 数据段的算法操作之外,还有一些操作,可以挑选数据通道上的某些独立的数据单元,并用这些单元组成新的数据集合。选择指令 SEL,可根据相关的 GE 标志位的值,独立地从源寄存器 Rn 或者 Rm 中选择一个 8 位数据。

```
SEL Rd, Rn, Rm
Rd[31 : 24] = GE[3] ? Rn[31 : 24] : Rm[31 : 24]
```

ARM 嵌入式系统开发

$$Rd[23 : 16] = GE[2] ? Rn[23 : 16] : Rm[23 : 16]$$

$$Rd[15 : 08] = GE[1] ? Rn[15 : 08] : Rm[15 : 08]$$

$$Rd[07 : 00] = GE[0] ? Rn[07 : 00] : Rm[07 : 00]$$

使用这些指令与其它的 SIMD 操作一起,可以非常有效地实现 Viterbi 算法的核心功能。Viterbi 算法是在通信系统中被广泛使用的用于符号恢复(symbol recovery)的一种算法,本来是一个统计学的最大可能性选择算法,也用于语音和手写识别引擎等领域。一般普遍认为 Viterbi 算法的核心是一个“加—比较—选择(ACS)”操作,实际上许多 DSP 处理器都特别定义了 ACS 指令。ARMv6 使用并行的(SIMD)加法、减法(可用于比较)和选择指令,可以实现一个特别有效的“加—比较—选择”操作:

```

ADD8    Rp1, Rs1, Rb1    ;path 1 = state1 + branch 1 (metric 更新)
ADD8    Rp2, Rs2, Rb2    ;path 2 = state2 + branch 2 (metric 更新)
USUB8   Rt, Rp1, Rp2     ;比较 metrics——设置 SIMD 标志
SEL      Rd, Rp2, Rp1     ;选择最好的(最小的)metric

```

在 ARM1136J-S 上,内核在 4 个通道上并行地运行 ACS 操作,总共只需 4 个周期,而在 ARMv5TE 指令集上,对同样的序列进行编码则必须串行地执行每一个操作,至少需要 16 个周期。由此可见,对于 8 位的 metrics* 操作,“加—比较—选择(ACS)”功能实现在 ARM1136J-S 上要快 4 倍。

15.1.2 打包指令

ARMv6 体系结构包括一组新的打包指令,如表 15.5 所列,使用一对取自不同源寄存器的 16 位数据来构造一个新的 32 位数据包。第 2 个操作数可以选择性地进行移位操作。打包指令对于成对的 16 位数值特别有用,这样就可以使用前面所描述的 16 位 SIMD 处理指令。

表 15.5 打包指令

指 令	描 述
PKHTB{<cond>} Rd, Rn, Rm {, ASR #<shift_imm>}	把 Rn 的高 16 位和移位的 Rm 的低 16 位打包保存到目标寄存器 Rd 中
PKHBT{<cond>} Rd, Rn, Rm {, LSL #<shift_imm>}	把移位的 Rm 的高 16 位和 Rn 的低 16 位打包保存到目标寄存器 Rd 中

* metric——度量,是已定义的测量方法和测量尺度[ISO/IEC 14598-1],泛指一个对象属性(或 2 个事物之间的差异)的量化表示。这里指实际接收数据与重构数据之间的匹配程度。——译者注

15.1.3 复数运算支持

复数运算(complex arithmetic)通常适用于通信信号处理,特别是在实现一些变换算法时,如在第8章中讲述的快速傅里叶变换(FFT)。在第8章中详细分析的许多实现细节,都和使用 ARMv4 或 ARMv5E 指令集实现复数乘法运算的效率有关。

ARMv6 增加了新的乘法指令,以提高复数乘法的速度,在表 15.6 中列出了这 2 条指令。如果指令中有后缀 X,则表示这些指令在执行乘法之前,首先要交换源操作数寄存器 Rs 中的 2 个 16 位半字数据。

表 15.6 支持 16 位复数乘法运算的指令

指 令	描 述
SMUAD{X}{<cond>} Rd, Rm, Rs	双 16 位有符号数的乘和加
SMUSD{X}{<cond>} Rd, Rm, Rs	双 16 位有符号数的乘和减

【例 15.1】 Ra 和 Rb 中分别包含 16 位系数的复数,其实部和虚部分别被打包存放在寄存器的低半部分和高半部分。

这里把 Ra 乘以 Rb,生成一个新的复数 Rc。假设这些 16 位的值都是采用 Q15 表示的小数,以下代码是针对 ARMv6 的:

```

SMUSD      Rt, Ra, Rb           ;实部 * 实部 - 虚部 * 虚部,结果为 Q30 表示
SMUADX     Rc, Ra, Rb           ;实部 * 虚部 + 虚部 * 实部,结果为 Q30 表示
QADD       Rt, Rt, Rt            ;转换为 Q31 & 饱和
QADD       Rc, Rc, Rc            ;转换为 Q31 & 饱和
PKHTB      Rc, Rc, Rt, ASR #16  ;把结果打包

```

和 ARMv5TE 上的实现比较:

```

SMULBB     Rc, Ra, Rb           ;实部 * 实部
SMULTT     Rt, Ra, Rb           ;虚部 * 虚部
QSUB       Rt, Rc, Rt            ;实部 * 实部 - 虚部 * 虚部,结果为 Q30 表示
SMULTB     Rc, Ra, Rb           ;虚部 * 实部
SMLABT     Rc, Rt, Rt            ;+ 实部 * 虚部,结果为 Q30 表示
QADD       Rt, Rt, Rt            ;转换为 Q31 & 饱和
QADD       Rc, Rc, Rc            ;转换为 Q31 & 饱和
MOV        Rc, Rc, LSR #16
MOV        Rt, Rt, LSR #16
ORR        Rt, Rt, Rc, LSL #16  ;把结果打包

```

以上代码的执行, ARMv5E 需要 10 个周期;而 ARMv6 只需 5 个周期。很明显,对于大量使用复数运算的算法,有了复数乘法运算的支持,可获得 2 倍的性能改进。

15.1.4 饱和指令

饱和运算最初被提出是在 ARMv5TE 体系结构的 E 扩展中,并在产品 ARM966E 和 ARM946E 中被引入。在 ARMv6 中增强了对饱和运算的支持,增加了单独的、更加灵活的、可以操作 32 位字和 16 位半字的饱和指令。除在表 15.7 中列出的这些指令,还有在 15.1.1 小节中描述的新的饱和运算 SIMD 操作。

表 15.7 饱和指令

指 令	描 述
SSAT Rd, #<BitPosition>, Rm, {<Shift>}	任意位置的有符号 32 位饱和。可以 LSL 和 ASR 移位
SSAT16{<cond>} Rd, #<immed>, Rm	在 2 个半长数据同一位置的双 16 位饱和
USAT Rd, #<BitPosition>, Rm, {<Shift>}	任意位置的无符号 32 位饱和。可以 LSL 和 ASR 移位
USAT16{<cond>} Rd, #<immed>, Rm	在 2 个半长数据同一位置的无符号双 16 位饱和

注意:在这些 32 位的饱和操作中,在饱和运算之前,可以对源寄存器 Rm 有选择地进行算术移位,这样就可以在同一个指令中进行数值范围的缩放。

15.1.5 绝对差值求和指令

这 2 条指令也许是 ARMv6 体系结构中最为与应用相关的——USAD8 和 USADA8。它们用于计算 8 位数据的绝对差值。这个运算在活动视频(motion video)压缩算法中特别有用,如 MPEG 或 H.263,包括移动预测算法(motion estimation algorithms)。其使用许多绝对差值求和(sum of absolute differences)操作来比较各个块的数值,从而实现对移动的计算(见图 15.2)。表 15.8 列出了这 2 个指令。

表 15.8 绝对差值求和

指 令	描 述
USAD8{<cond>} Rd, Rm, Rs	绝对差值求和
USADA8{<cond>} Rd, Rm, Rs, Rn	绝对差值累加求和

为了比较一个 $N \times N$ 的矩阵在 (x, y) 的图像 p_1 和图像 p_2 , 进行如下的绝对差值累加求和运算:

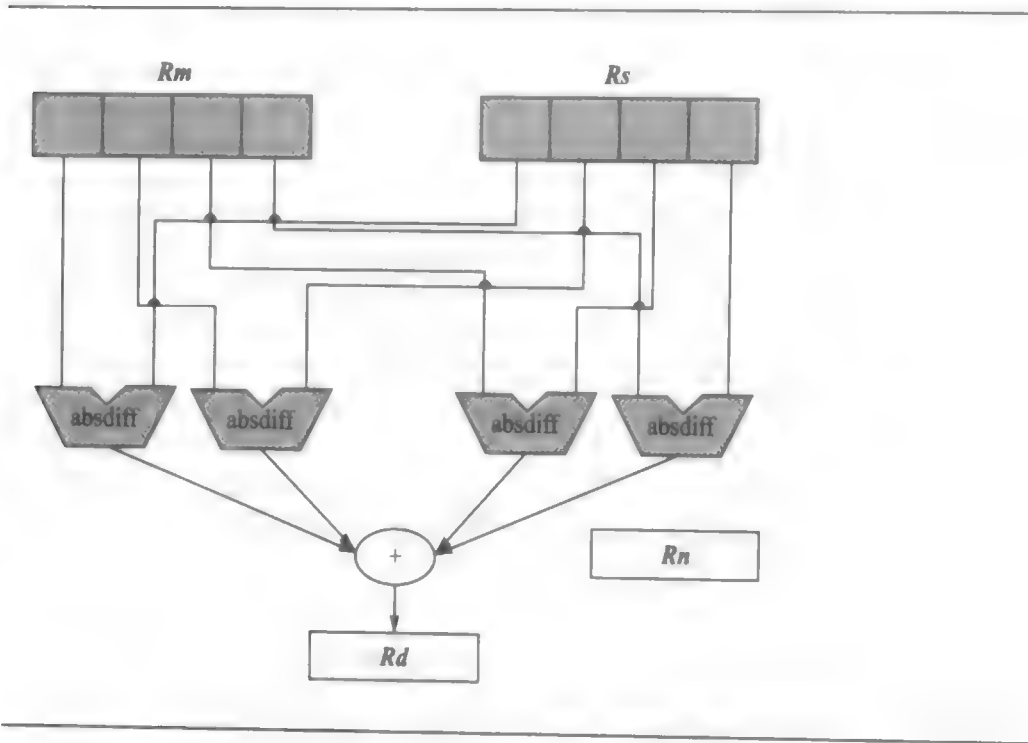


图 15.2 绝对差值求和运算

$$a(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} | p_1(x+i,y+j) - p_2(i,j) |$$

为了实现这个运算,可以使用这些新增的指令。使用下面的代码序列来计算 4 个像素的绝对差值的和:

```
LDR    p1, [p1Ptr], #4    ;从图像 p1 装载 4 个像素值
LDR    p2, [p2Ptr], #4    ;从图像 p2 装载 4 个像素值
;装载延迟
;装载延迟
USADA8  acc, p1, p2        ;累加计算绝对差值
```

8 位 SIMD 就有 4 倍性能的改善,再加上 USADA8 操作包含了累加求和操作,因此与一个基于 ARMv5TE 的实现相比,这种算法在性能上有了很大程度的改善。USAD8 操作也经常用于在计算出一个累加值之前完成进入循环的设置。

15.1.6 双 16 位乘法指令

虽然 ARMv5TE 已经对 ARM 引入了许多 DSP 特性,但在 ARMv6 中对 DSP 的支持更加深入。ARMv6 的实现(比如 ARM1136J)拥有双 16×16 位的乘法能力,已经可以和许多高端的专用 DSP 器件相比了。表 15.9 中列出了这些指令。

表 15.9 双 16 位乘法操作

指 令	描 述
SMLAD{X}{<cond>} Rd, Rm, Rs, Rn	双有符号乘累加, 32 位累加和
SMLALD{X}{<cond>} RdLo, RdHi, Rm, Rs	双有符号乘累加, 64 位累加和
SMLSD{X}{<cond>} Rd, Rm, Rs, Rn	双有符号乘减(从 32 位累加和中)
SMLSLD{X}{<cond>} RdLo, RdHi, Rm, Rs	双有符号乘减(从 64 位累加和中)

下面举例说明使用 SMLAD 作为有符号数的双重乘法用于一个点积计算的内循环:

```
MOV    R0, #0                ;累加器清零
Loop
  LDMIA R2!, {R4, R5, R6, R7} ;装载 8 个 16 位数据项
  LDMIA R1!, {R8, R9, R10, R11} ;装载 8 个 16 位系数
  SUBS  R3, R3, #8            ;从循环计数器中减去 8
  SMLAD R0, R4, R8, R0        ;4 次双乘累加
  SMLAD R0, R5, R9, R0
  SMLAD R0, R6, R10, R0
  SMLAD R0, R7, R11, R0
  BGT   Loop                  ;如果还有系数, 则循环
```

上面这个循环在 10 个周期内完成了 8 个 16×16 位数据项的乘累加, 而且没有使用任何的数据分块(data-blocking)技术。如果求点积的一组操作数都存放在寄存器中, 则其性能就接近于真正的单周期双乘累加操作了。

15.1.7 高位字乘法

ARMv5TE 中已经增加了很多算法操作, 这些操作在许多控制和通信应用的 DSP 算法中被广泛使用, 而且通常使用 Q15 数据格式。然而, 在音频处理应用中, 对于信号质量的描述, 16 位数据经常会不够。在这种情况下, 典型的解决方法就是使用 32 位数值。为此, ARMv6 增加了一些新的乘法指令, 用于操作 Q31 格式的数据(在第 8 章中详细描述了 Q 数据格式算法)。表 15.10 中列出了这些指令。

表 15.10 高位字 (most significant word) 乘法

指 令	描 述
SMMLA{R}{<cond>} Rd, Rm, Rs, Rn	有符号 32×32 乘法, 并把乘积的高 32 位累加到 32 位累加器 Rn
SMMLS{R}{<cond>} Rd, Rm, Rs, Rn	有符号 32×32 乘法, 从 Rn 左移 32 位(Rn << 32)减去乘积, 并获取结果的高 32 位
SMMUL{R}{<cond>} Rd, Rm, Rs	有符号数 32×32 乘法, 仅取乘积的高 32 位

指令中可选的 {R} 表示在乘高 32 位之前, 先在 64 位的乘积上加上固定的常数 0x80000000。这样就可以对结果进行有偏的舍入(biased rounding)。

15.1.8 密码算法乘法扩展

在一些密码算法中, 非常长的乘法运算十分常见。为了最大化对其处理吞吐量, 在现存的 32×32 乘法运算 UMULL 之上, 增加了一个新的 64+32×32→64 乘累加运算(见表 15.11)。

表 15.11 密码乘法 (cryptographic multiplication)

指 令	描 述
UMAAL{<cond>} RdLo, RdHi, Rm, Rs	特殊密码乘法 (RdHi; RdLo) = Rm * Rs + RdHi + RdLo

下面是一个使用这种新指令实现的非常高效的 64×64 位乘法的例子:

```
;输入:第 1 个 64 位乘数在(RaHi, RaLo)中
;第 2 个 64 位乘数在(RbHi, RbLo)中
umull64×64
    UMULL    R0, R2, RaLo, RbLo
    UMULL    R1, R3, RaHi, RbLo
    UMAAL    R1, R2, RaLo, RbHi
    UMAAL    R2, R3, RaHi, RbHi
;输出:128 位结果在(R3, R2, R1, R0)中
```

15.2 ARMv6 增加的系统和多处理器支持

随着系统变得越来越复杂,它们集中了多个处理器和处理引擎。这些引擎对存储器的使用可能不同,甚至采用不同的字节排列方式(大、小端)。为了支持这些系统中的相互通信,ARMv6 增加了对混合大小端系统、快速异常处理以及新的同步操作的支持。

15.2.1 混合大小端支持

传统的 ARM 体系结构有一个小端存储器系统或大端模式,可以在复位时进行切换。大端模式设置存储器系统按照大端格式组织指令和数据。

正如前面介绍的,ARM 内核经常被集成到复杂的混合大小端格式的片上系统中,这些系统在软件中经常需要同时处理小端和大端数据。ARMv6 新增了一个新的指令,以设置一段代码处理数据的字节排列方式(见表 15.12),另外还增加了一些单独的处理指令来提高在混合大小端环境下的处理效率。

表 15.12 设置端格式操作

指 令	描 述
SETEND <endian_specifier>	根据参数< endian_specifier >的值来改变默认的数据的端格式

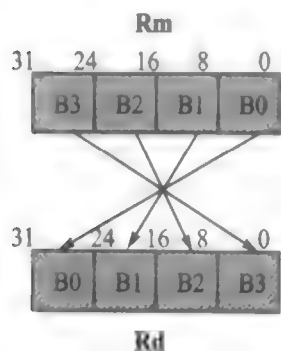
其中的 endian_specifier 可以是 BE(设置为大端格式),也可以是 LE(设置为小端格式)。典型地,当程序中存在相当数量的模块,在实现中需要各自特定的字节排列方式时,可以使用 SETEND 来设置。图 15.3 显示了单独的字节的处理指令。

15.2.2 异常处理

操作系统通常在堆栈中保存一次中断或异常的返回状态。ARMv6 增加了表 15.13 中的一些指令,以提高这种操作的效率。这种操作在中断/调度程序驱动系统中,出现频度是很高的。

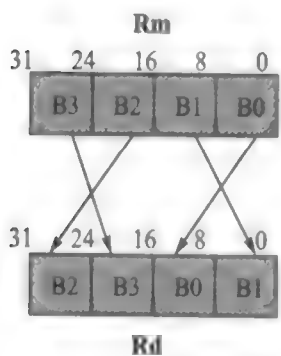
REV {<cond>} Rd, Rm

反转 32 位字中的所有 4 字节的顺序



REV16 {<cond>} Rd, Rm

反转 32 位字中的高低半字中 2 字节的顺序



REVSH {<cond>} Rd, Rm

反转有符号位的半字中的 2 字节的顺序

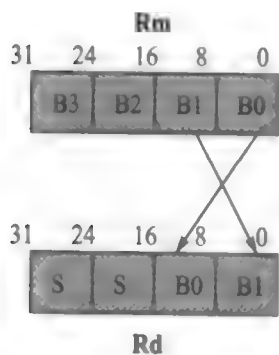


图 15.3 ARMv6 中的反转指令

表 15.13 异常处理

指 令	描 述
SRS<addressing_mode>, #mode {!}	保存返回状态(lr 和 spsr)在堆栈中,在特定的模式下,sp 指出堆栈地址
RFE< addressing_mode>, Rn {!}	从异常返回。根据 Rn 指出的地址,从堆栈中装载 pc 和 spsr
CPS<effect> <iflags> {, #<mode>}	改变处理器状态,开中断或关中断
CPS #<mode>	只改变处理器状态

15.2.3 多处理同步原语(Multiprocessing Synchronization Primitives)

由于片上系统 SoC 结构的复杂化,ARM 内核现在经常被用于有多个处理单元的设备,这些处理单元竞争使用系统的共享资源。在过去的 ARM 体系结构中,总是使用 SWP 指令来实现信号量(semaphores)在这种环境下的一致性。但是,随着片上系统 SoC 变得更加复杂,在某些情况下,SWP 指令的某些方面导致了一些性能瓶颈。因为 SWP 基本上是一个“阻塞”原语,它封锁了处理器的外部总线,并占用其大部分带宽,仅仅为了等待一个资源被释放。在这种意义上,SWP 指令通常被认为是“消极的”——任何运算都不能继续,直至资源被释放,SWP 操作返回。

为了解决这个问题,新指令 LDREX 和 STREX(装载和存储互斥的)被加入到了 ARMv6 结构中。表 15.14 中列出的这些指令,使用时非常简单、直接,它们是通过存储器系统内的一个系统监控器来实现的。LDREX 从存储器中装载一个值到寄存器,可以乐观地认为在处理这个数据时,不会有任何其它因素会改变该值。STREX 存储一个值到存储器,并且返回一个指示值,表示在最初的 LDREX 操作和这个写操作之间,存储器中的该值是否被修改过。采用这种方法,这些操作是“积极的”——可以继续处理用 LDREX 加载的数据,即使一些外部设备可能同时在修改这个值。只有真正发生了一个外部更新,该数值才会被丢弃和重载。

表 15.14 装载和存储互斥操作

指 令	描 述
LDREX{<cond>} Rd, [Rn]	从 Rn 所示的地址装载数据并且设置存储器监控器
STREX{<cond>} Rd, Rm, [Rn]	写数据到 Rn 所示的地址, 设标志 Rd(若成功, 则 Rd=0)

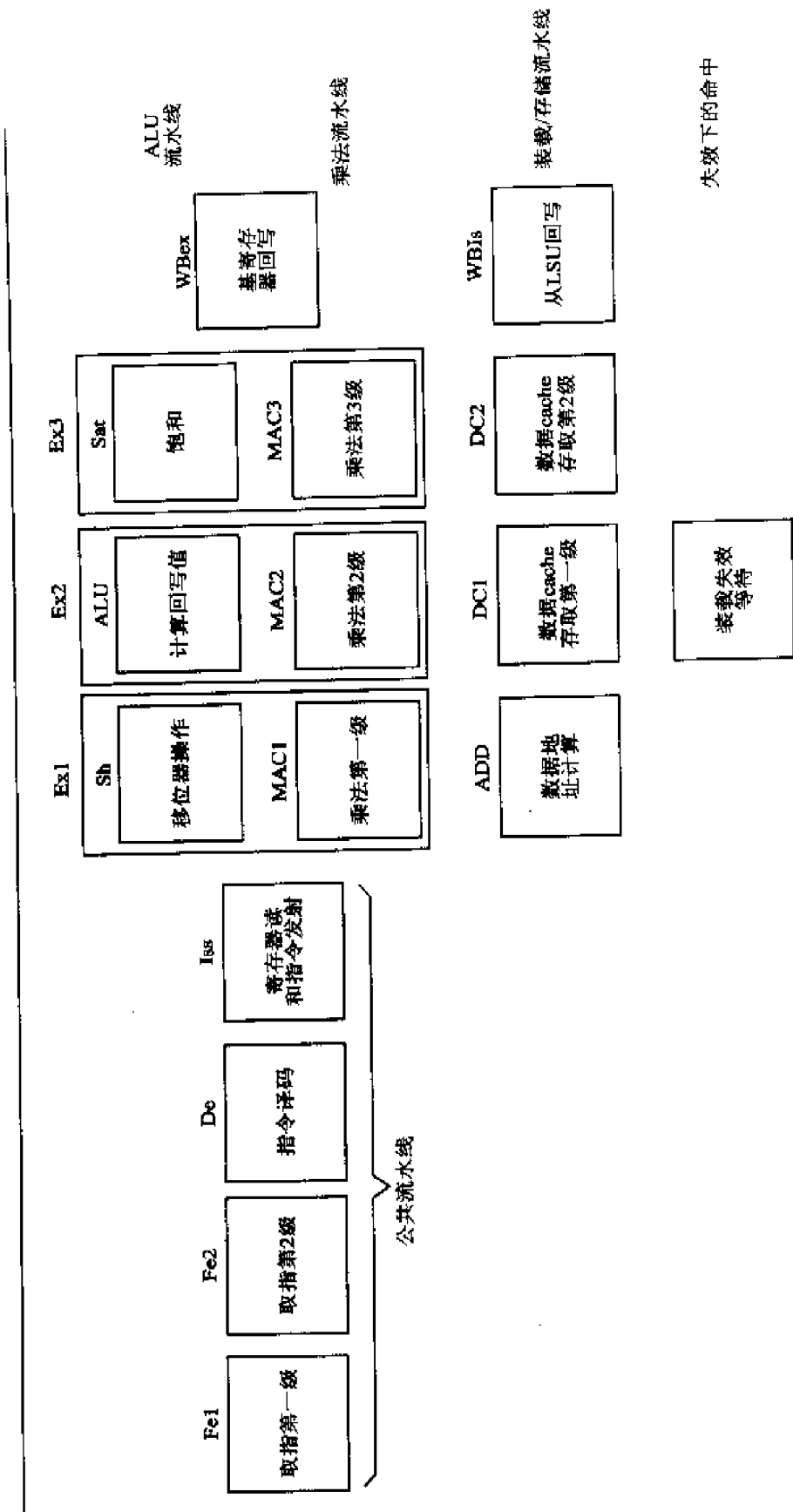
这种系统的最大特点就在于, 处理器不用再占用系统总线来等待一个将释放的信号量, 因此可以让出大部分的系统总线带宽, 供其它处理或处理器使用。

15.3 ARMv6 的实现

ARM 在 2002 年 12 月完成了 ARM1136J 的开发, 而且在编写本书同时, 使用这个内核的产品也正在设计中。到目前为止, ARM1136J 的流水线在各种 ARM 实现中是最复杂的。如图 15.4 所示, 它有一条 8 级流水线, 并带有独立、并行的数据装载/存储流水线和乘法/累加流水线。

并行的数据装载/存储单元(LSU)具有失效下的命中(hit-under-miss)能力。它允许在较慢的存储器系统正在完成装载或存储操作时, 装载和存储操作可以继续从流水线被发射(issued)并执行。通过把执行流水线 and 数据装载/存储操作分离, 内核可以获得可观的额外性能, 因为存储器系统通常要比内核的速度慢很多。失效下的命中把这种分离进一步扩展到了 L1-L2(一级-二级)存储器接口, 使得当一个 L1 cache 访问失效, L2 传输被执行时, 其它的 L1 命中可继续进行。

在微结构(microarchitecture)中的另一个大的改变就是, 从虚拟标记 cache 变为物理标记 cache。ARM 传统上使用虚拟标记的 cache, 即 MMU 位于 cache 和外部的二级存储器系统(L2)之间。在 ARMv6 中, 这点改变了, MMU 位于内核和一级存储器(L1)cache 之间, 使得所有的 cache 访问都使用物理地址(已经转换过的)。这种改进带来的大的好处之一就是, 当 ARM 运行大型操作系统时, 大大减少了在上下文切换时的 cache 刷新。另外, cache 刷新直接意味着更多的外存访问, 所以刷新的减少同样也会降低最终系统的功耗。在某些情况下, 这个体系结构上的改变可望带来 20% 以上的性能改善。



源自：ARM公司，ARM1136J技术参考手册，2003

图 15.4 ARM1136J 流水线

15.4 ARMv6 之后的未来技术

在 2003 年,ARM 发布了一些未来技术预告,其中包括 TrustZone 和 Thumb-2。这些技术都是非常新的,到写本书为止,它们正在被应用于一些新的微处理器内核中。下面将简要介绍这些新技术。

15.4.1 TrustZone

TrustZone 是一种体系结构的扩展,以传输的安全性为目标,可能会用于手机等电子消费类产品。将来或许会用于在线传输,实现下载音乐或视频等。2003 年 10 月,ARM 在发布 ARM1176JZ-S 时,第一次介绍了这种技术。

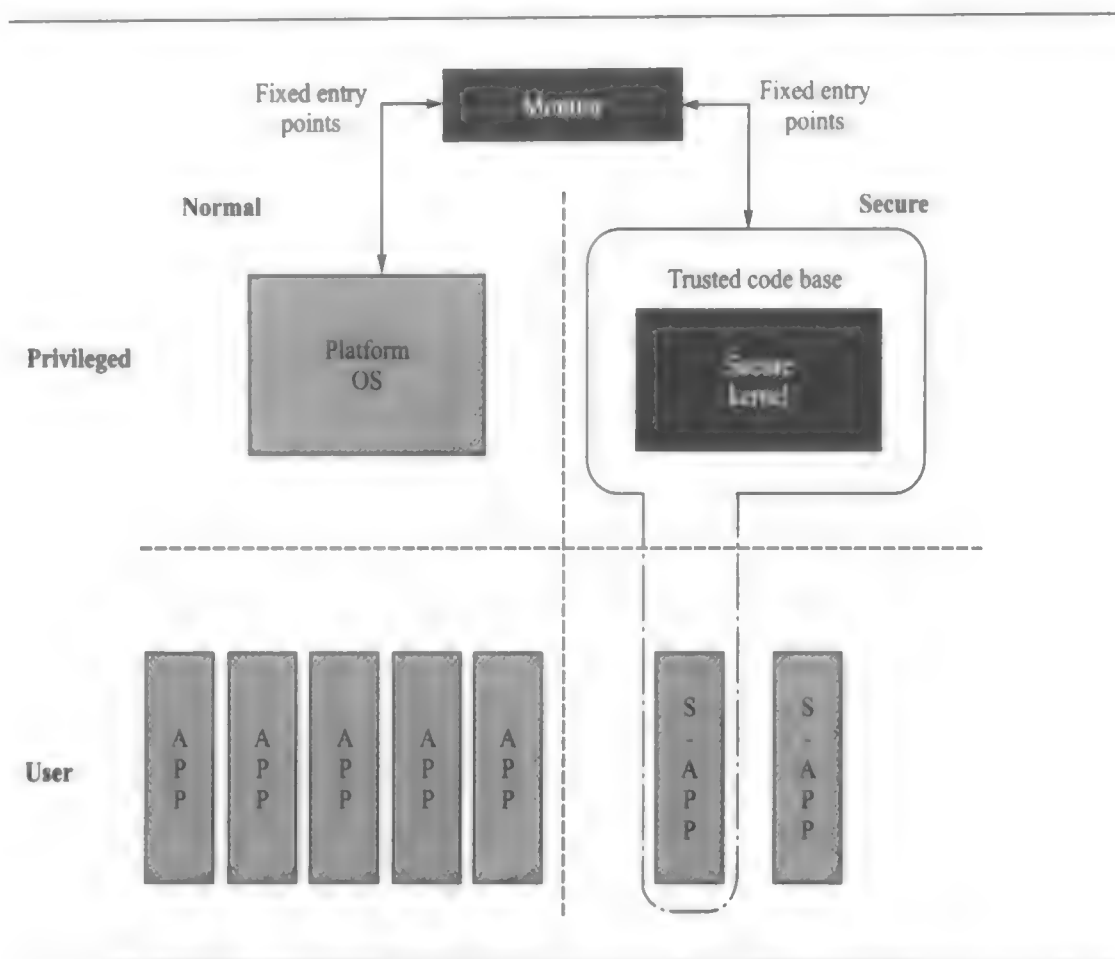
其基本思想是,现在的操作系统十分复杂(即使是在嵌入式设备上),以至于在软件中很难验证安全性和正确性。ARM 解决这个问题的办法是,添加新的操作“状态”到系统结构中,那里只有一个小的可验证的软件内核在运行,它将为更大的操作系统提供服务。这个微处理器内核通过在总线接口上的一些新的输出信号,可以控制系统外设只有在安全的“状态”下才有效。系统状态如图 15.5 所示。

TrustZone 在一些需要实现内容下载的设备中特别有用,比如手机以及其它带有网络连接的便携式设备。在编写本书时,这种结构的详细内容还没有公开。

15.4.2 Thumb-2

Thumb-2 是一种体系结构的扩展,是为了在高的代码密度上来改善性能。它允许混合 32 位的类 ARM(*ARM-like*)指令和 16 位 Thumb 指令。这种组合使得既可以获得 Thumb 指令的代码密度优势,又具有 32 位指令访问的性能优势。

Thumb-2 是在 2003 年 10 月发布的,将在 ARM1156T2-S 中被实现。在编写本书时,这种结构的详细内容还没有公开。



源自: Richard York, 一个 CPU 系统安全的新基础: ARM 体系结构的安全性扩展, 2003

图 15.5 采用 TrustZone 技术改进的安全结构

15.5 总 结

ARM 的体系结构不是一直不变的, 而是为了满足当今的电子消费产品设备的应用需要在不断发展和改进的。尽管 ARMv5TE 体系结构通过增加 DSP 支持, 获得了很大的成功, 而 ARMv6 体系结构不但进一步扩充了对 DSP 的支持, 同时还增加了对大型多处理器系统的支持。表 15.15 列出了这些新的技术及其所对应的不同处理器内核。

表 15.5 最新发布的内核

处理器内核	体系结构版本
ARM1136J-S	ARMv6J
ARM1156T2-S	ARMv6+Thumb-2
ARM1176JZ-S	ARMv6J+TrustZone

ARM 仍然专注于一个关键的因素——代码密度——并且在最近发布了在普通 Thumb 结构上的 Thumb-2 扩展。另外,新的热点——关于安全性的 TrustZone 技术也使得 ARM 在该领域中处于领先地位。

我们期待着未来 ARM 会有更多的创新!

附录 A

ARM 和 Thumb 汇编指令

- 如何使用这篇附录
- 语 法
- 按字母顺序列出 ARM 和 Thumb 指令
- ARM 汇编速查
- GNU 汇编快速查询

A ARM 和 Thumb 汇编指令

本附录列出了 ARM 和 Thumb 指令,并包含刚刚发布的 ARM 体系结构 ARMv6 指令。为了便于参考,将指令以字母为序进行排列。在 A.4 和 A.5 两节给出了 ARM 和 GNU 的汇编器 `armasm` 和 `gas` 指令快速查找索引。

本附录用于为实际编程提供指导,包括汇编代码的编写和对反汇编的输出进行解释。本附录的编写目的并不是提供 ARM 全面体系结构的参考,所以没有列出每一条指令位映射编码和操作行为的所有细节。关于这方面的详细内容,可以参考由 David Seal 编写,Addison Wesley 出版的 *ARM Architecture Reference Manual*。在附录 B 中将给出对 ARM 和 Thumb 指令设置编码的摘要。

A.1 如何使用这篇附录

在本附录中介绍每条指令时,都会列举所有可用的指令格式。例如,对于指令 `ADD` 的介绍,将会有:

`ADD<cond>{S} Rd, Rn, #<rotated_immed>` ARMv1

`<cond>` 和 `<rotated_immed>` 是在 A.2 节中将会介绍的 2 个标准域。`Rd` 和 `Rn` 表示 ARM 寄存器。这条指令只有在条件 `<cond>` 满足时,才能被执行。该附录会分别描述每条指令的具体执行情况。

`{S}` 表示可选的指令后缀。最右边的一列 ARMv1 表示可以使用该指令的最早版本。表 A.1 列出了它可能的版本。

表 A.1 指令类型

类 型	含 义
ARMvX	32 位 ARM 指令第一次出现在 ARM 体系中的第 X 版本
THUMBvX	16 位 Thumb 指令第一次出现在 Thumb 体系中的第 X 版本
MACRO	伪指令

注意: ARM 和 Thumb 的体系结构版本号并没有直接的相互联系。如 THUMBv1 使用在 ARMv4T 处理器中,THUMBv2 使用在 ARMv5T 处理器中,THUMBv3 使用在 ARMv6 处理器中。

每一条指令定义后都有一个注释段,用于描述使用该指令的限制条件。如果使用限制语句“`Rd must not be pc`”,则意味着该指令的功能只有在满足这个条件后才能实现。如果条件不满足,则这条指令执行后的结果可能是无法预测的,或者是可以预测的非正确结果(对此本附录不做详细的描述)。编程时不应超越这个条件限制。

A.2 语 法

在本附录中将使用如下的语法和缩写。

A.2.1 可选表达式

- $\{ \langle \text{expr} \rangle \}$ 是一个可选表达式。例如 LDR{B} 表示该指令可以是 LDR, 也可以是 LDRB。
- $\{ \langle \text{expr1} \rangle / \langle \text{expr2} \rangle / \dots / \langle \text{exprN} \rangle \}$ 至少包含一个分隔符“|”, 它是一系列的表达式。这一系列的表达式中至少出现其中的一条, 例如 LDR{B|H} 是 LDRB 或 LDRH 的缩写, 它不包含 LDR。如果想包含 LDR, 则应该写为: LDR{ |B|H}。

A.2.2 寄存器

- $Rd, Rn, Rm, Rs, RdHi, RdLo$ $r0 \sim r15$ 任一 ARM 寄存器。
- Ld, Ln, Lm, Ls $r0 \sim r7$ 的任意一个低编号的 ARM 寄存器。
- Hd, Hn, Hm, Hs $r8 \sim r15$ 的任意一个高编号的 ARM 寄存器。
- Cd, Cn, Cm $c0 \sim c15$ 的任意一个协处理寄存器。
- sp, lr 和 pc 各自代表 $r13, r14$ 和 $r15$ 。
- $Rn[a]$ 寄存器 Rn 位 a 值。因此 $Rn[a] = (Rn \gg a) \& 1$ 。
- $Rn[a:b]$ Rn 中位 $a \sim b$ 共 $(a+1-b)$ 位所构成的值。
- $RdHi, RdLo$ 一个 64 位数, $RdHi$ 是高 32 位, $RdLo$ 是低 32 位。

A.2.3 立即数

- $\langle \text{immedN} \rangle$ 一个 N 位的无符号立即数。例如 $\langle \text{immed8} \rangle$ 代表 $0 \sim 255$ 的任一整型数。 $\langle \text{immed5} \rangle * 4$ 可以表示数列 $0, 4, 8, 16, \dots, 124$ 中的任一数。
- $\langle \text{addrsN} \rangle$ 一个偏移量, 可以是地址或标注。这个地址必须满足 $pc - 2^N \leq \text{address} < pc + 2^N$ 。其中 pc 表示一个地址, 在 ARM 状态下等于当前指令地址加 8; 在 Thumb 状态下等于当前指令地址加 4。如果是一条 ARM 指令, 则该地址必须为 4 字节对齐的; 如果是一条 Thumb 指令, 则该地址必须为 2 字节对齐的。
- $\langle A-B \rangle$ $A \sim B$ 之间的任意整型数。
- $\langle \text{rotated_immed} \rangle$ 一个 32 位立即数, 它可以通过一个 8 位无符号数作左(或右)

偶数位循环移位而得到。也就是说： $\langle \text{rotated_immed} \rangle = \langle \text{immed8} \rangle \text{ ROR } (2 * \langle \text{immed4} \rangle)$ 。例如 0xff, 0x104, 0xe0000005, 0x0bc00000 都可以是一个 $\langle \text{rotated_immed} \rangle$ 数, 而 0x101 和 0x102 都不是。当使用一个循环立即数时, $\langle \text{shifter_C} \rangle$ 按照表 A. 3(将在第 A. 2. 5 小节中讨论)来设置。一个非零的循环移位可能导致进位标志的变化。所以可以通过 $\langle \text{immed8} \rangle$ 和 $2 * \langle \text{immed4} \rangle$ 明确地指定循环情况。

A. 2. 4 条件和标志

- $\langle \text{cond} \rangle$ 任意的标准 ARM 条件码。表 A. 2 列出了 $\langle \text{cond} \rangle$ 可能的值。
- $\langle \text{SignedOverflow} \rangle$ 一个标志, 表示在算术操作运算中产生了一个有符号数的溢出。例如, $0x7fffffff + 1 = 0x80000000$ 就产生了一个有符号数溢出, 因为 2 个 32 位有符号正数相加的结果变成了一个 32 位的负数。在 cpsr 中, 用 V 标志位记录有符号数溢出。
- $\langle \text{UnsignedOverflow} \rangle$ 一个标志, 表示在算术操作运算中产生了一个无符号数的溢出。例如, $0xffffffff + 1 = 0$ 就产生了一个 32 位无符号数溢出。在 cpsr 中, 用 C 标志位记录无符号数溢出。
- $\langle \text{NoUnsignedOverflow} \rangle$ 等于 $1 - \langle \text{UnsignedOverflow} \rangle$ 。
- $\langle \text{Zero} \rangle$ 一个标志, 表示一个算术运算或者逻辑运算的结果为 0。在 cpsr 中, 用 Z 标志位来标志这种情况。
- $\langle \text{Negative} \rangle$ 一个标志, 表示一个算术运算或者逻辑运算结果为负。也就是说, $\langle \text{Negative} \rangle$ 是结果的位 31。在 cpsr 中, 用 N 标志位来记录这种情况。

表 A. 2 ARM 条件助记符

$\langle \text{cond} \rangle$	执行该指令的条件	cpsr 状态
{AL}	始终执行	TRUE
EQ	相等(最后结果为 0)	$Z == 1$
NE	不等(最后结果非 0)	$Z == 0$
{CS HS}	进位位置位, 无符号大于或等于(用于比较后面)	$C == 1$
{CC LO}	进位位清零, 无符号小于(用于比较后面)	$C == 0$
MI	减(最后结果为负)	$N == 1$
PL	加(最后结果为正或 0)	$N == 0$
VS	V 标志位置位(结果产生有符号溢出)	$V == 1$
VC	V 标志位清零(结果没产生有符号溢出)	$V == 0$

续表 A.2

<cond>	执行该指令的条件	cpsr 状态
HI	无符号大于(用于比较后面)	$C=1 \& \& Z=0$
LS	无符号小于或等于(用于比较后面)	$C=0 Z=1$
GE	有符号大于或等于	$N=V$
LT	有符号小于	$N!=V$
GT	有符号大于	$N=V \& \& Z=0$
LE	有符号小于或等于	$N!=V Z=1$
NV	从不执行(只用于 ARMv1 和 ARMv2)不要使用	FALSE

A.2.5 移位操作

- **<imm_shift>** 一个通过立即数指定的移位位数。可能的移位方式有: LSL #<0-31>, LSR #<1-32>, ASR #<1-32>, ROR #<1-31> 和 RRX。表 A.3 给出了每一个移位的具体操作。
- **<reg_shift>** 一个通过寄存器指定的移位位数。可能的移位方式有: LSL Rs, LSR Rs, ASR Rs 和 ROR Rs。Rs 一定不能是 pc。Rs 的最低 8 位就是表 A.3 中所指的移位值 k 。Rs[31:8] 是被忽略的。
- **<shift>** **<imm_shift>** 或者 **<reg_shift>**。
- **<shifted_Rm>** Rm 为按照指定的移位操作移位后的值。见表 A.3。
- **<shifter_C>** 移位输出后的进位位的值。见表 A.3。

表 A.3 不同的移位类型的移位方法

Shift	k 的范围	<shifted_Rm>	<shifter_C>
LSL k	$k=0$	Rm	C(来自 cpsr)
LSL k	$1 \leq k \leq 31$	$Rm \ll k$	$Rm[32-k]$
LSL k	$k=32$	0	$Rm[0]$
LSL k	$k \geq 33$	0	0
LSR k	$k=0$	Rm	C
LSR k	$1 \leq k \leq 31$	(unsigned) $Rm \gg k$	$Rm[k-1]$
LSR k	$k=32$	0	$Rm[31]$
LSR k	$k \geq 33$	0	0
ASR k	$k=0$	Rm	C

续表 A.3

Shift	k 的范围	$\langle \text{shifted_Rm} \rangle$	$\langle \text{shifter_C} \rangle$
ASR k	$1 \leq k \leq 31$	$(\text{signed})\text{Rm} \gg k$	$\text{Rm}[k-1]$
ASR k	$k \geq 32$	$\text{Rm}[31]$	$\text{Rm}[31]$
ROR k	$k=0$	Rm	C
ROR k	$1 \leq k \leq 31$	$((\text{unsigned})\text{Rm} \gg k) (\text{Rm} \ll (32-k))$	$\text{Rm}[k-1]$
ROR k	$k \geq 32$	$\text{Rm ROR } (k \& 31)$	$\text{Rm}[(k-1) \& 31]$
RRX		$(\text{C} \ll 31) ((\text{unsigned})\text{Rm} \gg 1)$	$\text{Rm}[0]$

A.3 按字母顺序列出 ARM 和 Thumb 指令

指令将按字母顺序给出;但是如果同一个操作可以是有符号和无符号 2 种变量,那么这个主条目将以有符号变量为序来排列。

ADC 带进位位的 32 位数加法指令

- | | |
|--|---------|
| ① $\text{ADC} \langle \text{cond} \rangle \{S\} \text{ Rd, Rn, } \# \langle \text{rotated_immed} \rangle$ | ARMv1 |
| ② $\text{ADC} \langle \text{cond} \rangle \{S\} \text{ Rd, Rn, Rm } \{, \langle \text{shift} \rangle \}$ | ARMv1 |
| ③ $\text{ADC} \quad \text{Ld, Lm}$ | THUMBv1 |

Action

对 cpsr 的影响

- | | |
|--|-------------------|
| ① $\text{Rd} = \text{Rn} + \langle \text{rotated_immed} \rangle + \text{C}$ | 如果带后缀 S, 则更新 cpsr |
| ② $\text{Rd} = \text{Rn} + \langle \text{shifted_Rm} \rangle + \text{C}$ | 如果带后缀 S, 则更新 cpsr |
| ③ $\text{Ld} = \text{Ld} + \text{Lm} + \text{C}$ | 更新(见下面的注释) |

注释:

- 如果操作要更新 cpsr, 并且 Rd 不是 pc, 则 $\text{N} = \langle \text{Negative} \rangle$, $\text{Z} = \langle \text{Zero} \rangle$, $\text{C} = \langle \text{UnsignedOverflow} \rangle$, $\text{V} = \langle \text{SignedOverflow} \rangle$ 。
- 如果 Rd 是 pc, 则这条指令的执行会改变 pc 值, 下一条指令将跳转到新的 pc 值所指向的地址处。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 这样 cpsr 的值设置成 spsr 的值。
- 如果 Rn 或者 Rm 是 pc, 则它所用的值是当前指令的地址加上 8 字节。

例如:

ADDS	r0, r0, r2	; 两 64 位数的低 32 位相加
ADC	r1, r1, r3	; 两 64 位数的高 32 位相加
ADCS	r0, r0, r0	; r0 左移, 插入进位位 (RLX)

ADD

32 位数加法指令

① ADD<cond>S	Rd, Rn, #<rotated_immed>	ARMv1
② ADD<cond>S	Rd, Rn, Rm {, <shift>}	ARMv1
③ ADD	Ld, Ln, #<immed3>	THUMBv1
④ ADD	Ld, #<immed8>	THUMBv1
⑤ ADD	Ld, Ln, Lm	THUMBv1
⑥ ADD	Hd, Lm	THUMBv1
⑦ ADD	Ld, Hm	THUMBv1
⑧ ADD	Hd, Hm	THUMBv1
⑨ ADD	Ld, pc, #<immed8> * 4	THUMBv1
⑩ ADD	Ld, sp, #<immed8> * 4	THUMBv1
⑪ ADD	sp, #<immed7> * 4	THUMBv1

Action**对 cpsr 的影响**

① Rd = Rn + <rotated_immed>	如果带后缀 S, 则更新 cpsr
② Rd = Rn + <shifted_Rm>	如果带后缀 S, 则更新 cpsr
③ Ld = Ln + <immed3>	更新(见下面的注释)
④ Ld = Ld + <immed8>	更新(见下面的注释)
⑤ Ld = Ln + Lm	更新(见下面的注释)
⑥ Hd = Hd + Lm	受保护
⑦ Ld = Ld + Hm	受保护
⑧ Hd = Hd + Hm	受保护
⑨ Ld = pc + 4 * <immed8>	受保护
⑩ Ld = sp + 4 * <immed8>	受保护
⑪ sp = sp + 4 * <immed7>	受保护

注释:

- 如果操作要更新 cpsr, 并且 Rd 不是 pc, 则 N = <Negative>, Z = <Zero>, C = <UnsignedOverflow>, V = <SignedOverflow>。
- 如果 Rd 或者 Hd 是 pc, 则这条指令的执行会改变 pc 值, 下一条指令将跳转到新的 pc 值所指向的地址处。如果操作要更新 cpsr, 则处理器模式一

定要有一个 `spsr`, 这样 `cpsr` 的值设置成 `spsr` 的值。

- 如果 `Rn` 或者 `Rm` 是 `pc`, 则它所用的值是当前指令的地址加上 8 字节。
- 如果 `Hd` 或者 `Hm` 是 `pc`, 则它所用的值是当前指令的地址加上 4 字节。

例如:

```

ADD    r0, r1, #4           ; r0 = r1 + 4
ADDS   r0, r2, r2           ; r0 = r2 + r2 更新标志位
ADD    r0, r0, r0, LSL #1   ; r0 = 3 * r0
ADD    pc, pc, r0, LSL #2   ; 跳过 r0 + 1 条指令
ADD    r0, r1, r2, ROR r3   ; r0 = r1 + ((r2 >> r3) | (r2 << (32 - r3)))
ADDS   pc, lr, #4          ; 指令跳到 lr + 4 处, 存储 cpsr

```

ADR 小范围地址读取伪指令

ADR{L}<cond> Rd, <address> MACRO

这不是一条 ARM 指令, 而是一条宏汇编指令。它将基于 `pc` 的地址值读取到寄存器 `Rd` 中。ADR 伪指令被编译器替换成一条合适的 ARM 或者 Thumb 指令。ADRL 伪指令被编译器替换成 2 条 ARM 指令, 所以它可以比 ADR 读取到更大范围的地址。如果编译器不能产生一个指令序列来达到这个地址, 那么它就会产生一个错误。

下面的例子演示了如何调用 `r9` 所指向的函数。这里用 ADR 来设置 `lr` 为返回地址。这样, 它将汇编成 `ADD lr, pc, #4`。这里 `pc` 被看作当前指令地址加 8:

```

ADR    lr, return_address   ; 设置返回地址
MOV    r0, #0               ; 设置一个函数参数
BX     r9                   ; 调用这个函数
return_address              ; 恢复

```

AND 32 位数逻辑按位“与”运算指令

- | | | |
|----------------|--------------------------|---------|
| ① AND<cond>{S} | Rd, Rn, #<rotated_immed> | ARMv1 |
| ② AND<cond>{S} | Rd, Rn, Rm {, <shift>} | ARMv1 |
| ③ AND | Ld, Lm | THUMBv1 |

Action 对 cpsr 的影响

- ① $Rd = Rn \& \text{<rotated_immed>}$ 如果带后缀 S, 则更新 cpsr

- ② $Rd = Rn \& \langle \text{shifted_Rm} \rangle$ 如果带后缀 S, 则更新 cpsr
 ③ $Ld = Ld \& Lm$ 更新(见下面的注释)

注释:

- 如果操作要更新 cpsr, 并且 Rd 不是 pc, 则 $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{shift_C} \rangle$ (见表 A. 3), V 是受保护的。
- 如果 Rd 是 pc, 则这条指令的执行会改变 pc 值, 下一条指令将跳转到新的 pc 值所指向的地址处。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 这样 cpsr 的值设置成 spsr 的值。
- 如果 Rn 或者 Rm 是 pc, 则它所用的值是当前指令的地址加上 8 字节。

例如:

```
AND    r0, r0, #0xFF      ;提取最低 8 位
ANDS   r0, r0, #1 << 31   ;提取符号位
```

ASR Thumb 的算术右移指令(见 ARM 的 MOV 指令介绍)

- ① $ASR\ Ld, Lm, \# \langle \text{immed5} \rangle$ THUMBv1
 ② $ASR\ Ld, Ls$ THUMBv1

Action 对 cpsr 的影响

- ① $Ld = Lm\ ASR\ \# \langle \text{immed5} \rangle$ 更新(见注释)
 ② $Ld = Ld\ ASR\ Ls[7:0]$ 更 新

注释:

cpsr 更新为: $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{shift_C} \rangle$ (见表 A. 3)。

B 跳转指令

- ① $B \langle \text{cond} \rangle \langle \text{adderss25} \rangle$ ARMv1
 ② $B \langle \text{cond} \rangle \langle \text{adderss8} \rangle$ THUMBv1
 ③ $B \quad \quad \langle \text{adderss11} \rangle$ THUMBv1

跳转到指定的地址或标号。这个地址是一个偏移量。

例如:

```
B      label      ;无条件地跳转到 label 标号处
BGT    loop       ;有条件地继续一个循环
```


BIC 逻辑按位清零(AND NOT)指令

- ① BIC<cond>{S} Rd, Rn, #<rotated_immed> ARMv1
 ② BIC<cond>{S} Rd, Rn, Rm {, <shift>} ARMv1
 ③ BIC Ld, Lm THUMBv1

Action

对 cpsr 的影响

- ① $Rd = Rn \& \sim \text{rotated_immed}$ 如果带后缀 S, 则更新 cpsr
 ② $Rd = Rn \& \sim \text{shifted_Rm}$ 如果带后缀 S, 则更新 cpsr
 ③ $Ld = Ld \& \sim Lm$ 更新(见注释)

注释:

- 如果操作更新了 cpsr 且 Rd 不是 pc, 那么 N = <Negative>, Z = <Zero>, C = <shifter_C> (见表 A.3), V 是受保护的。
- 如果 Rd 是 pc, 则这条指令的执行会改变 pc 值, 下一条指令将跳转到新的 pc 值所指向的地址处。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 这样 cpsr 的值设置成 spsr 的值。
- 如果 Rn 或者 Rm 是 pc, 则它所用的值是当前指令的地址加上 8 字节。

例如:

BIC r0, r0, #1 << 22 ; 对 r0 的第 22 位清零

BKPT 断点指令

- ① BKPT <immed16> ARMv5
 ② BKPT <immed8> THUMBv2

除非开发调试工具硬件的屏蔽, 断点指令将会产生一个预取数据的异常中断。ARM 将会忽略这个立即数。但是这个立即数能够用来记录类似于断点号的调试信息。

BL 带链接的跳转指令(子程序调用)

- ① BL<cond> <address25> ARMv1
 ② BL <address22> THUMBv1

Action

对 cpsr 的影响

- ① $lr = ret + 0; pc = \text{address25}$ 无影响
 ② $lr = ret + 1; pc = \text{address22}$ 无影响

ARM 嵌入式系统开发

注释:

这些指令将设置寄存器 `lr` 的值为下一条指令的地址 `ret` 加上当前 `cpsr` 的 `T` 标志位的值。因此可以在子程序返回时,通过使用 `BX lr` 指令来恢复执行地址和 ARM 或者 Thumb 的状态。

例如:

<code>BL</code>	<code>subroutine</code>	;调用子程序(通过 <code>MOV pc, lr</code> 返回)
<code>BLVS</code>	<code>overflow</code>	;如果溢出,则调用子程序 <code>overflow</code>

BLX

带状态切换和链接的跳转指令(可能有状态切换的子程序调用)

- | | | |
|--------------------------------|--------------------------------|---------|
| ① <code>BLX</code> | <code><address25></code> | ARMv5 |
| ② <code>BLX<cond></code> | <code>Rm</code> | ARMv5 |
| ③ <code>BLX</code> | <code><address22></code> | THUMBv2 |
| ④ <code>BLX</code> | <code>Rm</code> | THUMBv2 |

Action

对 `cpsr` 的影响

- | | |
|--|-----------------------------------|
| ① <code>lr = ret + 0; pc = <address25></code> | <code>T = 1</code> (切换为 Thumb 状态) |
| ② <code>lr = ret + 0; pc = Rm & 0xffffffe</code> | <code>T = Rm & 1</code> |
| ③ <code>lr = ret + 1; pc = <address22></code> | <code>T = 0</code> (切换到 ARM 状态) |
| ④ <code>lr = ret + 1; pc = Rm & 0xffffffe</code> | <code>T = Rm & 1</code> |

注释:

- 这些指令将设置寄存器 `lr` 的值为下一条指令的地址 `ret` 加上当前 `cpsr` 的 `T` 标志位的值。因此可以在子程序返回时,通过使用 `BX lr` 指令来恢复执行地址和 ARM 或者 Thumb 的状态。
- `Rm` 一定不能为 `pc`。
- `Rm & 3` 一定不能为 2,因为这将跳转到一个没有对齐的 ARM 指令。

例如:

<code>BLX thumb_code</code>	;在 ARM 状态下调用一个 Thumb 子程序
<code>BLX r0</code>	;调用一个由 <code>r0</code> 所指的子程序
	;如果 <code>r0</code> 是偶数,则为 ARM 指令;如果是奇数,则为 Thumb 指令

BX 带状态切换的跳转指令(可能导致状态切换的跳转)

BXJ

① BX<cond>	Rm	ARMv4T
② BX	Rm	THUMBv1
③ BXJ<cond>	Rm	ARMv5J

Action	对 cpsr 的影响
① pc = Rm & 0xffffffe	T = Rm & 1
② pc = Rm & 0xffffffe	T = Rm & 1
③ 依赖 JE 的设置位	J, T 位会受到影响

注释:

- 如果 Rm 是 pc 而且指令是字对齐的,则 Rm 的值为当前指令地址加 8 字节 (ARM 状态下)或者加 4 字节(Thumb 状态下)。
- Rm & 3 一定不能等于 2,因为这将导致跳转到一个没有对齐的 ARM 指令。
- 如果 JE(Java Enable)设置位被清除,则 BXJ 相当于 BX 指令;否则,指令的执行情况将由 Java 的扩展硬件体系结构所定义。典型地,它设置 cpsr 的 J=1,并通过一个 Java 程序计数器 jpc 的通用寄存器来开始执行 Java 指令。

例如:

```

BX    lr        ;从 ARM 或者 Thumb 子程序返回
BX    r0        ;跳转到 r0 指定的 ARM 或 Thumb 函数

```

CDP 协处理器数据操作指令

① CDP<cond>	<copro>, <op1>, Cd, Cn, Cm, <op2>	ARMv2
② CDP2	<copro>, <op1>, Cd, Cn, Cm, <op2>	ARMv5

这些指令初始化一个与协处理器相关的操作。<copro>为 p0~p15 的一个协处理器号。如果系统没有协处理器,则将产生一个未定义指令的陷阱。<op1>和<op2>是协处理器的操作码,协处理器寄存器 Cd, Cn 和 Cm 由协处理器解释,ARM 将忽略它们。CDP2 提供一组附加的协处理器指令。

CLZ 前导零计数指令

CLZ<cond>	Rd, Rm	ARMv5
-----------	--------	-------

ARM 嵌入式系统开发

Rd 被设置为 Rm 左移而不产生无符号溢出的最大位数。等价地,它也是 Rm 的二进制数的最高位的 0 的个数。如果 $Rm = 0$, 则 Rn 等于 32。下面的例子规格化 r0 的值, 这样 r0 的位 31 被置位。

```
CLZ    r1, r0           ;找出规格化的移位数
MOV    r0, r0, LSL r1   ;规格化,r0 的第 31 位被置位(if r0!=0)
```

CMN

取负比较指令

- | | | |
|-------------|----------------------|---------|
| ① CMN<cond> | Rn, #<rotated_immed> | ARMv1 |
| ② CMN<cond> | Rn, Rm {, <shift>} | ARMv1 |
| ③ CMN | Ln, Lm | THUMBv1 |

Action:

- ① cpsr 标志通过 $(Rn + \text{<rotated_immed>})$ 的结果来设置。
- ② cpsr 标志通过 $(Rn + \text{<shifted_Rm>})$ 的结果来设置。
- ③ cpsr 标志通过 $(Ln + Lm)$ 的结果来设置。

注释:

- 在 cpsr 中: N = <Negative>, Z = <Zero>, C = <Unsigned-Overflow>, V = <SignedOverflow>。这些和 CMP 指令把第 2 个操作数取相反数产生的效果相同。
- 如果 Rn 或者 Rm 是 pc, 则这个值是该指令的地址加 8 字节。

例如:

```
CMN    r0, #3           ;r0 和 -3 进行比较
BLT    label            ;if(r0<-3) goto label
```

CMP

2 个 32 位整型数的比较指令

- | | | |
|-------------|----------------------|---------|
| ① CMP<cond> | Rn, #<rotated_immed> | ARMv1 |
| ② CMP<cond> | Rn, Rm {, <shift>} | ARMv1 |
| ③ CMP | Ln, #<immed8> | THUMBv1 |
| ④ CMP | Rn, Rm | THUMBv1 |

Action:

- ① cpsr 标志通过 $(Rn - \text{<rotated_immed>})$ 的结果来设置。
- ② cpsr 标志通过 $(Rn - \text{<shift_Rm>})$ 的结果来设置。
- ③ cpsr 标志通过 $(Ln - \text{<immed8>})$ 的结果来设置。

④ cpsr 标志通过(Rn-Rm)的结果来设置。

注释:

- 在 cpsr 中: N = <Negative>, Z = <Zero>, C = <NoUnsigned-Overflow>, V = <SignedOverflow>。进位位之所以按照这样设置,是因为减法 $x-y$ 相当于加法 $x+\sim y+1$ 。如果 $x+\sim y+1$ 溢出,则进位位等于 1。当 $x\geq y$ (也就是 $x-y$ 不会溢出)时,就是这种情况。
- 如果 Rn 或者 Rm 是 pc,则这个值是当前指令地址加 8 字节(ARM 指令)或者加 4 字节(Thumb 指令)。

例如:

```

CMP    r0, r1, LSR#2      ;r0 和 r1/4 作比较
BHS    label              ;if(r0 >= (r1/4)) goto label;
```

CPS 改变处理器的状态,更改 cpsr 的被选择位

- | | |
|-----------------------------|---------|
| ① CPS #<mode> | ARMv6 |
| ② CPSID <flags> {, #<mode>} | ARMv6 |
| ③ CPSIE <flags> {, #<mode>} | ARMv6 |
| ④ CPSID <flags> | THUMBv3 |
| ⑤ CPSIE <flags> | THUMBv3 |

Action;

- ① $\text{cpsr}[4:0] = \text{<mode>}$
- ② $\text{cpsr} = \text{cpsr} | \text{mask}; \{ \text{cpsr}[4:0] = \text{<mode>} \}$
- ③ $\text{cpsr} = \text{cpsr} \& \sim \text{mask}; \{ \text{cpsr}[4:0] = \text{<mode>} \}$
- ④ $\text{cpsr} = \text{cpsr} | \text{mask}$
- ⑤ $\text{cpsr} = \text{cpsr} \& \sim \text{mask}$

通过<flags>的字母标识来确定要设置的掩码的哪些位(见表 A.4)。中断禁止变量 ID(Interrupt Disable)通过对 cpsr 中断屏蔽位置 1 来关闭中断。中断允许变量 IE(Interrupt Enable)通过对 cpsr 中断屏蔽位清零来开启中断。

表 A.4 CPS 的 flags 字母

字 母	对 cpsr 产生影响的位	Bit set in mask
a	不精确的数据中止屏蔽位	$0x100 = 1 \ll 8$
i	中断屏蔽位	$0x080 = 1 \ll 7$
f	快速中断屏蔽位	$0x040 = 1 \ll 6$

CPY 复制指令,复制 ARM 的一个寄存器到另一个寄存器,而不影响 cpsr

- | | | |
|-------------|--------|---------|
| ① CPY<cond> | Rd, Rm | ARMv6 |
| ② CPY | Rd, Rm | THUMBv3 |

除了在 Thumb 指令下 Rd 和 Rm 都是 r0~r7 的低地址寄存器这种情况,这条指令被汇编成 MOV<cond> Rd, Rm。这样它是一个新的设置 Rd = Rm,而不影响 cpsr 的操作指令。

EOR 32 位数的逻辑“异或”操作指令

- | | | |
|----------------|--------------------------|---------|
| ① EOR<cond>{S} | Rd, Rn, #<rotated_immed> | ARMv1 |
| ② EOR<cond>{S} | Rd, Rm { , <shift> } | ARMv1 |
| ③ EOR | Ld, Lm | THUMBv1 |

Action

对 cpsr 的影响

- | | |
|--|------------------|
| ① $Rd = Rn \wedge \text{<rotated_immed>}$ | 如果带后缀 S,则更新 cpsr |
| ② $Rd = Rn \wedge \text{<shifted_Rm>}$ | 如果带后缀 S,则更新 cpsr |
| ③ $Ld = Ld \wedge Lm$ | 更新(见下面的注释) |

注释:

- 如果操作更新了 cpsr 而且 Rd 不是 pc,则 N = <Negative>, Z = <Zero>, C = <shifter_C>(见表 A.3), V 是受保护的。
- 如果 Rd 是 pc,则这条指令的执行会改变 pc 值,下一条指令将跳转到新的 pc 值所指向的地址处。如果操作要更新 cpsr,则处理器模式一定要有一个 spsr,这样 cpsr 的值设置成 spsr 的值。
- 如果 Rn 或者 Rm 是 pc,则计算出来的值等于这条指令的地址加 8 字节。

例如:

EOR r0,r0,#1 << 16

;r0 的位 16 取反

LDC

一个或多个字的协处理器数据读取指令

- ① LDC<cond>{L} <copro>, Cd,[Rn{, #{-}<immed8>*4}](!) ARMv2
- ② LDC<cond>{L} <copro>, Cd,[Rn], #{-}<immed8>*4 ARMv2
- ③ LDC<cond>{L} <copro>, Cd,[Rn], <option> ARMv2
- ④ LDC2{L} <copro>, Cd,[Rn{, #{-}<immed8>*4}](!) ARMv5
- ⑤ LDC2{L} <copro>, Cd,[Rn], #{-}<immed8>*4 ARMv5
- ⑥ LDC2{L} <copro>, Cd,[Rn], <option> ARMv5

指令从存储器读取数据到指定的协处理器。<copro>是 p0~p15 的协处理器编号。如果协处理器不存在,则将会产生一个未定义指令陷阱。存储器通过地址递增,从一系列连续的存储器单元字中读取数据。初始地址通过表 A.5 中的寻址方式来说明。协处理器控制从存储器中读取的字的数量,最多不超过 16 个字。域{L}和 Cd 由协处理器解释,ARM 将忽略它。典型的情况下,Cd 申明协处理器的目的寄存器。域<option>是一个 8 位的整型数,它的解释依赖于具体的协处理器。

表 A.5 LDC 的寻址模式

寻址模式	访问地址	返回给 Rn 的值
[Rn{, #{-}<immed>}]	$Rn + \{-\}<immed>$	Rn 受保护的
[Rn{, #{-}<immed>}]!	$Rn + \{-\}<immed>$	$Rn + \{-\}<immed>$
[Rn], #{-}<immed>	Rn	$Rn + \{-\}<immed>$
[Rn], <option>	Rn	Rn 受保护的

如果地址不是 4 的倍数,则地址是没有对齐的。指令对于没对齐的地址的操作与 LDM 一样。

LDM

从存储器里读取一组 32 位字到 ARM 寄存器指令

- ① LDM<cond><amode> Rn(!),<register_list>{^} ARMv1
- ② LDMIA Rn!,<register_list> THUMBv1

指令从连续的存储器单元中装载多个字到寄存器中。<register_list>由花括号{}列出一系列的需要装载的寄存器。尽管指令允许以任意的顺序列出寄存器,但是顺序是不保存在指令中的。由于通常的存储器读取顺序是

按照序号从小到大的,所以应将寄存器的顺序按照从小到大排列。

下面的这段伪代码展示了通常情况下 LDM 的指令执行情况。使用 $\langle \text{register_list} \rangle[i]$ 表示出现在寄存器列表中的第 i 个寄存器,0 表示起始寄存器。假定列表按照寄存器号增加的顺序排列。

```

N = 寄存器列表  $\langle \text{register\_list} \rangle$  中的寄存器数目
start = 表 A.6 中的最低访问地址
for(  $i = 0; i < N; i++$  )
     $\langle \text{register\_list} \rangle[i] = \text{memory}(\text{start} + i * 4, 4);$ 
if (指定了 '!') then 按照表 A.6 更新 Rn

```

表 A.6 LDM 的寻址模式

寻址模式	最低地址	最高地址	如果指定了“!”,则返回 Rn 的值
{IA FD}	Rn	$Rn + N * 4 - 4$	$Rn + N * 4$
{IB ED}	$Rn + 4$	$Rn + N * 4$	$Rn + N * 4$
{DA FA}	$Rn - N * 4 + 4$	Rn	$Rn - N * 4$
{DB EA}	$Rn - N * 4$	$Rn - 4$	$Rn - N * 4$

$\text{memory}(a, 4)$ 根据当前处理器的大小端情况,返回从地址 a 开始的 4 字节的数据。如果地址 a 不等于 4 的倍数,则装载是没有对齐的。因为没有对齐的装载指令要依赖具体的体系结构版本、存储器系统和协处理器系统 (CP15) 的配置,所以应尽量避免不对齐的装载操作。如果外部存储器系统不会因为不对齐的装载操作而异常中止,则通常会遵照下面的规则:

- 如果内核有一个系统协处理器,而且 CP15:c1:c0:0 的位 1 (A_bit) 或者位 22 (U_bit) 是 1,则不对齐的批量存储器读取将会产生一个不对齐数据异常中断。
- 否则,操作将会忽略最低的 2 地址位。

表 A.6 列出了由 $\langle \text{amode} \rangle$ 说明的几种寻址模式。如果 Rn 后指定了后缀“!”,则基地址寄存器按照表 A.6 更新;否则它将是被保护的。

注意:最低编号的寄存器通常是读最低存储器地址的数据。

寻址模式的前一半分别代表 IA (过后增加, Increment After)、IB (预先增加, Increment Before)、DA (过后减少, Decrement After)、DB (预先减少, Decrement Before)。在增加模式下,顺序地向前存储寄存器列表的寄存器,

过后增加从地址 R_n 开始,预先增加从地址 R_n+4 开始。在减少模式下,如果采用向后存储寄存器列表的寄存器,并且按照存储器地址减少的方式读取数据,则和增加模式有一样的效果。但是起始地址不一样,过后减少是 R_n ,预先减少是 R_n-4 。

寻址模式的后一半代表栈的类型,是满堆栈还是空堆栈,是递增栈(升序栈)还是下降栈(降序栈):FD(满降序,Full Descending),ED(空降序,Empty Descending),FA(满升序,Full Ascending),EA(空升序,Empty Ascending)。对于一个满堆栈, R_n 指向栈的最后一个值;对于空堆栈, R_n 指向栈的第一个未使用的位置。ARM 的栈通常是下降式满堆栈(满降序)。可以根据自己的偏好来选择使用满降序还是空升序,LDC 指令同样适合这些寻址模式。

注释:

- 对于 Thumb(第 2 条指令格式), R_n 和寄存器列表中的寄存器必须是 $r0 \sim r7$ 。
- 列表中的寄存器数目 N 必须是非 0 的。
- R_n 不能是 pc 。
- 如果指定了“!”(回写),则 R_n 不能出现在寄存器列表中。
- 如果 pc 出现在寄存器列表中,则在 ARMv5 或以上体系结构中,处理器执行一条 BX 指令,跳转到要装载的地址。如果是 ARMv4 或以下的体系结构,则处理器直接跳转到要装载的地址。
- 如果 <register_list> 后面带符号“-”,则操作将会发生变化。这时的处理器模式一定不能是用户模式或系统模式。如果 pc 不在寄存器列表中,则出现在寄存器列表中的寄存器参照寄存器的用户模式版本,而且一定不能指定回写;如果 pc 在寄存器列表中,则在标准操作的基础上,还要将 $spsr$ 的值拷贝到 $cpsr$ 中去。
- 存储器读取的时序依赖于指令的执行情况。在使用多字节存取指令处理 I/O 时,要注意时序问题。如果与时序有关,则要检查 I/O 映射的存储器位置是否在页表内。不要超出页表边界,而且不要让 pc 在寄存器列表中。

例如:

```

LDMIA    r4!, {r0, r1}      ; r0 = *r4, r1 = *(r4+4), r4 += 8
LDMDB    r4!, {r0, r1}      ; r1 = *(r4-4), r0 = *(r4-8), r4 -= 8
LDMEQFD  sp!, {r0, pc}      ; if(result zero) then unstack r0, pc

```

LDMFD	sp, {sp}^	;load sp_usr from sp_current
LDMFD	sp, {r0 - pc}^	;return from exception, restore cpsr

LDR 从存储器到寄存器的单一数据读取指令

- ① LDR<cond>{B} Rd, [Rn {, #{-}<immed12>}]{!} ARMv1
- ② LDR<cond>{B} Rd, [Rn, {-}Rm {, <imm_shift>}]{!} ARMv1
- ③ LDR<cond>{B}{T} Rd, [Rn], #{-}<immed12> ARMv1
- ④ LDR<cond>{B}{T} Rd, [Rn], {-}Rm {, <imm_shift>} ARMv1
- ⑤ LDR<cond>{H|SB|SH} Rd, [Rn], {, #{-}<immed8>}]{!} ARMv4
- ⑥ LDR<cond>{H|SB|SH} Rd, [Rn, {-}Rm]{!} ARMv4
- ⑦ LDR<cond>{H|SB|SH} Rd, [Rn], #{-}<immed8> ARMv4
- ⑧ LDR<cond>{H|SB|SH} Rd, [Rn], {-}Rm ARMv4
- ⑨ LDR<cond>D Rd, [Rn, {, #{-}<immed8>}]{!} ARMv5E
- ⑩ LDR<cond>D Rd, [Rn, {-}Rm]{!} ARMv5E
- ⑪ LDR<cond>D Rd, [Rn, #{-}<immed8> ARMv5E
- ⑫ LDR<cond>D Rd, [Rn], {-}Rm ARMv5E
- ⑬ LDREX<cond> Rd, [Rn] ARMv6
- ⑭ LDR{B|H} Ld, [Ln, #<immed5> * <size>] THUMBv1
- ⑮ LDR{B|H|SB|SH} Ld, [Ln, Lm] THUMBv1
- ⑯ LDR Ld, [pc, #<immed8> * 4] THUMBv1
- ⑰ LDR Ld, [sp, #<immed8> * 4] THUMBv1
- ⑱ LDR<cond><type>Rd, <label> MACRO
- ⑲ LDR<cond> Rd, =<32-bit-value> MACRO

第 1~17 条指令都是装载单一的数据,操作码后缀指定数据类型,使用前变址或后变址的寻址模式。表 A.7 和表 A.8 列出了不同的寻址模式和数据类型。

表 A.7 LDR 寻址模式

寻址模式	入口地址 a	返回给 Rn 的值
[Rn {, #{-}<immed>}]	Rn + {{-}<immed>}	Rn 受保护的
[Rn {, #{-}<immed>}]{!}	Rn + {{-}<immed>}	Rn + {{-}<immed>}
[Rn, {-}Rm {, <shift>}]	Rn + {-}<shifted_Rm>	Rn 受保护的
[Rn, {-}Rm {, <shift>}]{!}	Rn + {-}<shifted_Rm>	Rn + {-}<shifted_Rm>

续表 A.7

寻址模式	入口地址 a	返回给 Rn 的值
$[Rn], \# \{-\} \langle \text{immed} \rangle$	Rn	$Rn \mid \{-\} \langle \text{immed} \rangle$
$[Rn], \{-\} Rm \{, \langle \text{shift} \rangle\}$	Rn	$Rn \mid \{-\} \langle \text{shifted_Rm} \rangle$

表 A.8 LDR 的数据类型

Load	数据类型	$\langle \text{size} \rangle$ (bytes)	操 作
LDR	word	4	$Rd = \text{memory}(a, 4)$
LDRB	unsigned Byte	1	$Rd = (\text{zero-extend})\text{memory}(a, 1)$
LDRBT	Byte Translated	1	$Rd = (\text{zero-extend})\text{memoryT}(a, 1)$
LDRD	Double word	8	$Rd = \text{memory}(a, 4)$ $R(d+1) = \text{memory}(a+4, 4)$
LDREX	Word EXclusive	4	$Rd = \text{memoryEx}(a, 4)$
LDRH	unsigned Halfword	2	$Rd = (\text{zero-extend})\text{memory}(a, 2)$
LDRSB	Signed Byte	1	$Rd = (\text{sign-extend})\text{memory}(a, 1)$
LDRSB	Signed Halfword	2	$Rd = (\text{sign-extend})\text{memory}(a, 2)$
LDRT	word Translated	4	$Rd = \text{memoryT}(a, 4)$

在表 A.8 中, $\text{memory}(a, n)$ 从地址 a 开始顺序地读取 n 个字节的数据。数据的打包根据当前处理器的字节排列顺序不同而不同。函数 $\text{memoryT}(a, n)$ 操作和 $\text{memory}(a, n)$ 一样, 但它不考虑当前的处理器模式, 都按照特权级用户模式进行处理。函数 $\text{memoryEx}(a, n)$ 由 LDREX 独占调用执行。如果地址 a 有共享 TLB 属性, 则它将会把地址 a 作为当前处理器访问的独占标记, 而清除处理器的其它独占访问地址, 否则处理器将会记录这里有一个悬而未决的独占访问地址。独占性只会影响到 STREX 指令的执行。

如果地址 a 不是 $\langle \text{size} \rangle$ 的倍数, 则读取指令是没有对齐的。因为没对齐的读取操作依赖于系统体系结构的版本、存储器系统和协处理器系统 (CP15) 的配置, 所以尽量避免不对齐的操作。假定外部的存储系统不会因为不对齐读取数据而产生异常中断, 则通常会按照如下的规则执行。在该规则下, A 是协处理器寄存器 CP15: $c1, c0:0$ 的位 1, U 是协处理器寄存器 CP15: $c1, c0:0$ 的位 22。如果系统没有协处理器, 则 $A = U = 0$ 。

- 如果 $A = 1$, 则除了在 $U = 1$ 的字对齐的双字读取操作外, 都会因为不对齐的读取操作而产生一个数据对齐错误的异常中断。
- 如果 $A = 0, U = 1$, 则 $\text{LDR}\{\text{T}|\text{H}|\text{SH}\}$ 支持不对齐的读取操作, LDRD

支持字对齐的读取操作。一个非字对齐(non-word-aligned)的 LDRD 读取操作将会产生一个数据对齐错误的异常中断。

- 如果 $A = 0, U = 0$, 则 LDR 和 LDRT 返回值是 $\text{memory}(a \& \sim 3, 4) \text{ ROR}((a \& 3) * 8)$ 。所有的其它不对齐的操作结果都是不可预知的,但是不会产生一个对齐错误。

第 18 条指令通过由 $\langle \text{label} \rangle$ 指定的地址产生一个和 pc 相关的读取操作。也就是说,在支持这条指令的情况下,如果 $\langle \text{offset} \rangle = \langle \text{label} \rangle - \text{pc}$ 也在范围内,那么这条指令汇编成 $\text{LDR} \langle \text{cond} \rangle \langle \text{type} \rangle \text{ Rd}, [\text{pc}, \# \langle \text{offset} \rangle]$ 。

第 19 条指令产生一条指令,把给定的 32 位数值保存到寄存器 Rd。通常这条指令是 $\text{LDR} \langle \text{cond} \rangle \text{ Rd}, [\text{pc}, \# \langle \text{offset} \rangle]$, 这个 32 位数保存在以地址 $(\text{pc} + \langle \text{offset} \rangle)$ 开始的一个文字池(literal pool)中。

注释:

- 对于双字的读取操作(第 9~12 条), Rd 必须是 r0~r12 的偶数号寄存器。
- 如果寻址模式更新了 Rn, 则 Rd 和 Rn 一定要是不同的寄存器。
- 如果 Rd 是 pc, 则 $\langle \text{size} \rangle$ 一定要等于 4。对于 ARMv4 及其以前的体系结构来说,直接跳转到读取的地址;对于 ARMv5 或者更高的体系结构,则系统执行一条 BX 指令,跳转到读取的地址。
- 如果 Rn 是 pc, 则寻址模式一定不能更新 Rn。对于 ARM, Rn 的值等于当前指令的地址加 8 字节;对于 Thumb, Rn 的值等于当前指令的地址加 4 字节。
- Rm 一定不能是 pc。
- 对于 ARMv6, 使用 LDREX 和 STREX 来执行原语,而不用指令 SWP。

例如:

```

LDR    r0, [r0]           ;r0 = *(int *)r0;
LDRSH  r0, [r1], #4       ;r0 = *(short *)r1; r1 += 4;
LDRB   r0, [r1, #-8]!     ;r1 -= 8; r0 = *(char *)r1;
LDRD   r2, [r1]           ;r2 = *(int *)r1; r3 = *(int *) (r1 + 4);
LDRSB  r0, [r2, #55]      ;r0 = *(signed char *) (r2 + 55);
LDRCC  pc, [pc, r0, LSL #2] ;if(C==0) goto *(pc + 4 * r0);
LDRB   r0, [r1], -r2, LSL #8; r0 = *(char *)r1; r1 -= 256 * r2;
LDR    r0, =0x12345678     ;r0 = 0x12345678;

```

LSL

Thumb 的逻辑左移指令(参见 ARM 的 MOV 指令)

① LSL Ld, Lm, #<immed5>

THUMBv1

② LSL Ld, Ls

THUMBv1

Action

对 cpsr 的影响

① Ld = Lm LSL # <immed5>

更新(见下面的注释)

② Ld = Ld LSL Ls[7:0]

更新

注释:

cpsr 的更新为: N = <Negative>, Z = <Zero>, C = <shift_C> (见表 A.3)。

LSR

Thumb 的逻辑右移指令(参见 ARM 的 MOV 指令)

① LSR Ld, Lm, # <immed5>

THUMBv1

② LSR Ld, Ls

THUMBv1

Action

对 cpsr 的影响

① Ld = Lm LSR # <immed5>

更新(见下面的注释)

② Ld = Ld LSR Ls[7:0]

更新

注释:

cpsr 的更新为: N = <Negative>, Z = <Zero>, C = <shift_C> (见表 A.3)。

MCR

从一个 ARM 寄存器到协处理器的传送指令

MCRR

① MCR<cond> <copro>, <op1>, Rd, Cn, Cm {, <op2>} ARMv2

② MCR2 <copro>, <op1>, Rd, Cn, Cm {, <op2>} ARMv5

③ MCRR<cond> <copro>, <op1>, Rd, Rn, Cm ARMv5E

④ MCRR2 <copro>, <op1>, Rd, Rn, Cm ARMv6

这些指令把寄存器 Rd 的值传送到指定的协处理器去。第 3 和第 4 条指令还将传送第 2 个寄存器 Rn。<copro> 是 p0~p15 的协处理器号。如果没有协处理器存在,则会产生一个未定义指令的陷阱。由 <op1> 和 <op2> 指定协处理器操作,由 Cn 和 Cm 指定协处理器寄存器号,它们都由协处理器解释,ARM 将会忽略它们。Rd 和 Rn 一定不能是 pc。协处理器 p15 控制存储器管理操作,见第 13 章和第 14 章中对存储器管理单元 MPU 和 MMU 的描述。例如下面的代码序列进行初始化对齐错误的检查。

MRC p15, 0, r0, c1, c0, 0 ;read the MMU register, c1

ORR r0, r0, #2 ;set the A bit

ARM 嵌入式系统开发

MCR p15, 0, r0, c1, c0, 0 ;write the MMU register, c1

MLA 带累加的乘法指令

MLA<cond>{S} Rd, Rm, Rs, Rn ARMv2

Action 对 cpsr 的影响

$Rd = Rn + Rm * Rs$ 如果指定了后缀 S, 则更新

注释:

- Rd 作为低 32 位的目标寄存器。
- Rd, Rm, Rs 和 Rn 都不能是 pc。
- Rd 和 Rm 必须是不同的寄存器。
- 指令可能会因为操作数 Rs 的值而提前结束。Rs 应尽量使用很小的数或者常数, 见附录 D。
- 如果 cpsr 被更新, 则 N = <Negative>, Z = <Zero>, C 是不可预知的, V 是受保护的。因为使用 MLAS 经常会产生多余的指令周期而影响其它指令的执行, 所以应避免使用指令 MLAS。用 MLA 后面跟一条比较指令来代替, 这样就会避免因乘法结果带来的互锁。

MOV 把一个 32 位数传送到寄存器的传送指令

- | | | |
|----------------|----------------------|---------|
| ① MOV<cond>{S} | Rd, #<rotated_immed> | ARMv1 |
| ② MOV<cond>{S} | Rd, Rm {, <shift>} | ARMv1 |
| ③ MOV | Ld, #<immed8> | THUMBv1 |
| ④ MOV | Ld, Ln | THUMBv1 |
| ⑤ MOV | Hd, Lm | THUMBv1 |
| ⑥ MOV | Ld, Hm | THUMBv1 |
| ⑦ MOV | Hd, Hm | THUMBv1 |

Action

对 cpsr 的影响

- | | |
|----------------------------------|---------------------|
| ① $Rd = \text{<rotated_immed>}$ | 如果指定了后缀 S, 则更新 cpsr |
| ② $Rd = \text{<shifted_Rm>}$ | 如果指定了后缀 S, 则更新 cpsr |
| ③ $Ld = \text{<immed8>}$ | 更新 cpsr(见下面的注释) |
| ④ $Ld = Ln$ | 更新 cpsr(见下面的注释) |
| ⑤ $Hd = Lm$ | 受保护的 |
| ⑥ $Ld = Hm$ | 受保护的 |

A ARM 和 Thumb 汇编指令

⑦ Hd = Hm

受保护的

注释:

- 如果操作要更新 cpsr 而且 Rd 不是 pc, 则 N = <Negative>, Z = <Zero>, C = <shift_C> (见表 A. 3), V 是受保护的。
- 如果 Rd 或者 Hd 是 pc, 这条指令的计算结果改变 pc 值, 则下一条指令将跳转到以 pc 的新值为地址处。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 这样 cpsr 的值设置成 spsr 的值。
- 如果 Rm 是 pc, 则所使用的值是当前指令的地址加 8 字节。
- 如果 Hm 是 pc, 则所使用的值是当前指令的地址加 4 字节。

例如:

```
MOV    r0, #0x00ff0000    ;r0 = 0x00ff0000
MOV    r0, r1, LSL #2      ;r0 = 4 * r1
MOV    pc, lr              ;从子程序返回(pc = lr)
MOVS   pc, lr              ;从异常返回(pc = lr, cpsr = spsr)
```

MRC 从协处理器到 ARM 寄存器的传送指令**MRRC**

- ① MRC<cond> <copro>, <op1>, Rd, Cn, Cm, <op2> ARMv2
- ② MRC2 <copro>, <op1>, Rd, Cn, Cm, <op2> ARMv5
- ③ MRRC<cond> <copro>, <op1>, Rd, Rn, Cm ARMv5E
- ④ MRRC2 <copro>, <op1>, Rd, Rn, Cm ARMv6

这些指令把一个指定的协处理器的 32 位数传送到 ARM 寄存器 Rd。第 3 和第 4 条指令还要传送一个 32 位数到寄存器 Rn。<copro> 是 p0~p15 范围内的协处理器号。如果系统没有协处理器, 则将会产生一个未定义指令的陷阱。由 <op1> 和 <op2> 指定协处理器操作, 由 Cn 和 Cm 指定的协处理器寄存器号都由协处理器解释, ARM 将会忽略它们。对于第 1 和第 2 条指令, 如果 Rd 是 pc, 则 cpsr 的高 4 位(cpsr 的 NZCV 4 个条件标志位)被设置为被传送的 32 位数的最高 4 位, pc 不会受到影响。对于其它的 2 条指令, Rd 和 Rn 应是不同的寄存器, 而且不能是 pc。

协处理器 CP15 控制存储器管理操作(见第 12 章和第 13 章)。例如, 下面的指令将从 CP15 中读取主 ID 寄存器:

MRC pl5, 0, r0, c0, c0 ; read the MMU ID Register, c0

MRS 从状态寄存器(cpsr or spsr)到 ARM 寄存器的传送指令

- ① MRS<cond> Rd, cpsr ARMv3
 ② MRS<cond> Rd, spsr ARMv3

这些指令分别把 Rd 寄存器设置为 $Rd = cpsr$ 和 $Rd = spsr$ 。Rd 一定不能是 pc。

MSR 从 ARM 寄存器到状态寄存器(cpsr or spsr)的传送指令

- ① MSR<cond> cpsr_<fields>, #<rotated_immed> ARMv3
 ② MSR<cond> cpsr_<fields>, Rm ARMv3
 ③ MSR<cond> spsr_<fields>, #<rotated_immed> ARMv3
 ④ MSR<cond> spsr_<fields>, Rm ARMv3

Action:

- ① $cpsr = (cpsr \& \sim\langle mask \rangle) | (\langle rotated_immed \rangle \& \langle mask \rangle)$
 ② $cpsr = (cpsr \& \sim\langle mask \rangle) | (Rm \& \langle mask \rangle)$
 ③ $spsr = (spsr \& \sim\langle mask \rangle) | (\langle rotated_immed \rangle \& \langle mask \rangle)$
 ④ $spsr = (spsr \& \sim\langle mask \rangle) | (Rm \& \langle mask \rangle)$

这些指令通过掩码的值来改变 cpsr 或 spsr 的被选择字节的值。通过指定<fields>为一个或者几个字母的序列,来确定掩码<mask>被选择要置位的字节。见表 A.9。

表 A.9 指定的<fields>格式

<fields>字母	含 义	掩码(<mask>)位设置
c	控制字节(Control byte)	0x000000ff
x	扩展字节(eXtension byte)	0x0000ff00
s	状态字节(Status byte)	0x00ff0000
f	标志字节(Flags byte)	0xff000000

一些老的 ARM 工具包允许 cpsr 或者 cpsr_all 代替 cpsr_fsxc。它们也分别使用 cpsr_flag 和 cpsr_ctl 代替 cpsr_f 和 cpsr_c。spsr 也有相同的格式,然而这些格式都是很陈旧的,所以现在不使用它们。下面的代码可以切换到系统模式并且允许 IRQ 中断,这些对于可重入的中断服务程序是非常有

用的:

MRS	r0, cpsr	;读 cpsr 的状态
BIC	r0, r0, #0x9f	;清除 IRQ 禁止位和模式位
ORR	r0, r0, #0x1f	;设置为系统模式
MSR	cpsr_c, r0	;更新 cpsr 的控制字节

MUL

乘法指令

① MUL<cond>{S}	Rd, Rm, Rs	ARMv2
② MUL	Ld, Lm	THUMBv1

Action

对 cpsr 的影响

- ① $Rd = Rm * Rs$ 如果指定了后缀 S, 则更新
- ② $Ld = Lm * Ld$ 更新

注释:

- Rd 或 Ld 用于保存结果的低 32 位。
- Rd, Rm 和 Rs 都不能是 pc。
- Rd 和 Rm 必须是不同的寄存器, 同样 Ld 和 Lm 也应是不同的。
- 执行可能会因为操作数 Rs 或 Ld 的值而提前中止, 所以要尽量使 Rs 和 Ld 的值比较小或使用常数。
- 如果 cpsr 更新了, 则 N = <Negative>, Z = <Zero>, C 是不可预知的, V 是受保护的。因为使用 MULS 经常会产生多余的指令周期而影响其它指令的执行, 所以应避免使用指令 MULS。用 MUL 后面跟一条比较指令来代替, 这样就会避免因乘法结果带来的互锁。

MVN

将一个 32 位数逻辑取反后传送到寄存器的指令

① MVN<cond>{S}	Rd, #<rotated_immed>	ARMv1
② MVN<cond>{S}	Rd, Rm {, <shift>}	ARMv1
③ MVN	Ld, Lm	THUMBv1

Action

对 cpsr 的影响

- ① $Rd = \sim \langle \text{rotated_immed} \rangle$ 如果指定了后缀 S, 则更新
- ② $Rd = \sim \langle \text{shifted_Rm} \rangle$ 如果指定了后缀 S, 则更新
- ③ $Ld = \sim Lm$ 更新(见下面的注释)

- 如果操作要更新 cpsr 并且 Rd 不是 pc, 则 $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{shift_C} \rangle$ (参见表 A.3), V 是受保护的。
- 如果 Rd 是 pc, 则指令的执行结果将会影响到下一条指令的执行地址。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr 寄存器, 这样, cpsr 的值将被设置成 spsr 的值。
- 如果 Rm 是 pc, 则所使用的值是当前指令的地址加 8 字节。

```

MVN    r0, #0xff      ;r0 = 0xffffffff00
MVN    r0, #0          ;r0 = -1

```

在 Thumb 状态下求相反数的指令(在 ARM 状态下求相反数用 RSB 指令)

THUMBv1

对 cpsr 的影响

更新(见下面的注释)

- cpsr 被更新为: N=<Negative>, Z=<Zero>, C=<NoUnsignedOverflow>, V=<SignedOverflow>。

- 这和 ARM 状态下的 RSBS Ld, Lm, #0 指令有相同的操作。

空操作指令

MACRO

ORR

32 位数的按位逻辑“或”操作指令

① ORR<cond>{S} Rd, Rn, #<rotated_immed>

ARMv1

- ② ORR<cond>{S} Rd, Rn, Rm {, <shift>} ARMv1
 ③ ORR Ld, Lm THUMBv1

Action	对 cpsr 的影响
① $Rd = Rn \mid \langle \text{rotated_immed} \rangle$	如果指定了后缀 S, 则更新
② $Rd = Rn \mid \langle \text{shifted_Rm} \rangle$	如果指定了后缀 S, 则更新
③ $Ld = Ld \mid Lm$	更新(见下面的注释)

注释:

- 如果操作要更新 cpsr 并且 Rd 不是 pc, 则 $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{shifter_C} \rangle$ (见表 A.3), V 是受保护的。
- 如果 Rd 是 pc, 则指令的执行结果将会影响到下一条指令的执行地址。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr 寄存器。这样, cpsr 的值将被设置成 spsr 的值。
- 如果 Rn 或 Rm 是 pc, 则所使用的值是当前指令的地址加上 8 字节。

例如:

ORR r0, r0, #1 << 13 ;把 r0 的位 13 置 1

PKH 把 2 个 16 位的半字合并成一个 32 位字的操作指令

- ① PKHBT<cond> Rd, Rn, Rm {, LSL #<0-31>} ARMv6
 ② PKHTB<cond> Rd, Rn, Rm {, ASR #<1-32>} ARMv6

Action

- ① $Rd[15:00] = Rn[15:00]; Rd[31:16] = \langle \text{shifted_Rm} \rangle[31:16]$
 ② $Rd[31:16] = Rn[31:16]; Rd[15:00] = \langle \text{shifted_Rm} \rangle[15:00]$

注释:

Rd, Rn 和 Rm 都一定不能是 pc。cpsr 不会受到影响。

例如:

PKHBT r0, r1, r2, LSL #16 ;r0 = (r2[15:00] << 16) | r1[15:00]
 PKHTB r0, r2, r1, ASR #16 ;r0 = (r2[31:15] << 16) | r1[31:15]

PLD 预取暗示指令

- ① PLD [Rn {, #<-><immed12>}] ARMv5E
 ② PLD [Rn, {-}Rm {, <imm_shift>}] ARMv5E

Action:

- ① 从地址($Rn + \{-\} \langle \text{immed12} \rangle$)预取。
- ② 从地址($Rn + \{-\} \langle \text{shifted_Rm} \rangle$)预取。

这条指令不会影响到处理器寄存器(只是增加 pc 的值)。它仅仅暗示程序将来可能会到给定的地址去读数据。一个带高速缓存的处理器可能把这个当成从给定的地址读取数据到高速缓存中去的暗示。这条指令不会产生任何的数据异常或者其它的存储器系统错误。如果 Rn 是 pc, 则 Rn 的值等于当前指令的地址加上 8 字节。Rm 一定不能是 pc。

例如:

```
PLD    [r0, #7]           ;从 r0+7 预取数据
PLD    [r0, r1, LSL #2]   ;从 r0+4*r1 预取数据
```

POP Thumb 状态下,从堆栈中弹出多个字节到寄存器组(ARM 状态使用 LDM)

POP <register_list> THUMBv1

Action:

等价于 ARM 指令:LDMFD sp!, <register_list>。

<register_list>可以包含 r0~r7 的寄存器和 pc 寄存器。下面的代码恢复 ARM 的低编号寄存器并且从子程序中返回:

```
POP {r0-r7, pc}
```

PUSH 在 Thumb 状态下,保存多个寄存器数据到堆栈中去(ARM 状态下使用 STM)

PUSH <register_list> THUMBv1

Action:

等价于 ARM 指令:STMFD sp!, <register_list>。

<register_list>可以包含 r0~r7 的寄存器和 lr 寄存器。下面的代码保存 ARM 的低编号寄存器和链接寄存器:

```
PUSH {r0-r7, lr}
```

QADD
QDADD
QDSUB
QSUB

饱和的(Saturated) 有符号和无符号的算术运算指令

① QADD<cond>	Rd, Rm, Rn	ARMv5E
② QDADD<cond>	Rd, Rm, Rn	ARMv5E
③ QSUB<cond>	Rd, Rm, Rn	ARMv5E
④ QDSUB<cond>	Rd, Rm, Rn	ARMv5E
⑤ {U}QADD16<cond>	Rd, Rn, Rm	ARMv6
⑥ {U}QADDSUBX<cond>	Rd, Rn, Rm	ARMv6
⑦ {U}QSUBADDX<cond>	Rd, Rn, Rm	ARMv6
⑧ {U}QSUB16<cond>	Rd, Rn, Rm	ARMv6
⑨ {U}QADD8<cond>	Rd, Rn, Rm	ARMv6
⑩ {U}QSUB8<cond>	Rd, Rn, Rm	ARMv6

Action:

- ① $Rd = \text{sat32}(Rm + Rn)$
- ② $Rd = \text{sat32}(Rm + \text{sat32}(2 * Rn))$
- ③ $Rd = \text{sat32}(Rm - Rn)$
- ④ $Rd = \text{sat32}(Rm - \text{sat32}(2 * Rn))$
- ⑤ $Rd[31:16] = \text{sat16}(Rn[31:16] + Rm[31:16]);$
 $Rd[15:00] = \text{sat16}(Rn[15:00] + Rm[15:00])$
- ⑥ $Rd[31:16] = \text{sat16}(Rn[31:16] + Rm[31:16]);$
 $Rd[15:00] = \text{sat16}(Rn[15:00] - Rm[15:00])$
- ⑦ $Rd[31:16] = \text{sat16}(Rn[31:16] - Rm[31:16]);$
 $Rd[15:00] = \text{sat16}(Rn[15:00] + Rm[15:00])$
- ⑧ $Rd[31:16] = \text{sat16}(Rn[31:16] - Rm[31:16]);$
 $Rd[15:00] = \text{sat16}(Rn[15:00] - Rm[15:00])$
- ⑨ $Rd[31:24] = \text{sat8}(Rn[31:24] + Rm[31:24]);$
 $Rd[23:16] = \text{sat8}(Rn[23:16] + Rm[23:16]);$
 $Rd[15:08] = \text{sat8}(Rn[15:08] + Rm[15:08]);$
 $Rd[07:00] = \text{sat8}(Rn[07:00] + Rm[07:00])$
- ⑩ $Rd[31:24] = \text{sat8}(Rn[31:24] - Rm[31:24]);$
 $Rd[23:16] = \text{sat8}(Rn[23:16] - Rm[23:16]);$

$$\begin{aligned} \text{Rd}[15:08] &= \text{sat8}(\text{Rn}[15:08] - \text{Rm}[15:08]); \\ \text{Rd}[07:00] &= \text{sat8}(\text{Rn}[07:00] - \text{Rm}[07:00]) \end{aligned}$$

注释:

- 如果没有指定前缀 U, 则是有符号运算, 否则是无符号运算。如果是有符号, 则 $\text{satN}(x)$ 饱和 x 到范围 $-2^{N-1} \leq x < 2^{N-1}$; 如果是无符号, 则 $\text{satN}(x)$ 饱和 x 到范围 $0 \leq x < 2^N$ 。
- 如果饱和发生了, 则 cpsr 的 Q 标志位置 1; 否则它是受保护的。
- Rd, Rn, Rm 都不能是 pc。
- X 操作对于复数的打包运算很有用。下面的代码假定 bits[15:00] 是复数的实部, bits[31:16] 是虚部。

例如:

QDADD	r0, r0, r2	; 把 Q30 的值 r2 加到 Q31 的累加器 r0
QADD16	r0, r1, r2	; SIMD 饱和相加
QADDSUBX	r0, r1, r2	; 打包的复数运算 $r0 = r1 + i * r2$
QSUBADDX	r0, r1, r2	; 打包的复数运算 $r0 = r1 - i * r2$

REV

字或半字的字节反转指令

- | | | |
|---------------|--------|---------------|
| ① REV<cond> | Rd, Rm | ARMv6/THUMBv3 |
| ② REV16<cond> | Rd, Rm | ARMv6/THUMBv3 |
| ③ REVSH<cond> | Rd, Rm | ARMv6/THUMBv3 |

Action:

- ① $\text{Rd}[31:24] = \text{Rm}[07:00]; \text{Rd}[23:16] = \text{Rm}[15:08];$
 $\text{Rd}[15:08] = \text{Rm}[23:16]; \text{Rd}[07:00] = \text{Rm}[31:24]$
- ② $\text{Rd}[31:24] = \text{Rm}[23:16]; \text{Rd}[23:16] = \text{Rm}[31:24];$
 $\text{Rd}[15:08] = \text{Rm}[07:00]; \text{Rd}[07:00] = \text{Rm}[15:08]$
- ③ $\text{Rd}[31:08] = \text{sign-extend}(\text{Rm}[07:00]); \text{Rd}[07:00] = \text{Rm}[15:08]$

注释:

- Rd 和 Rm 一定不能是 pc。
- Thumb 状态下, Rd 和 Rm 必须在 r0~r7 的范围内, 不能指定条件 <cond>。
- 这些指令对于大端数据和小端数据的相互转化非常有用。

例如:

REV	r0, r0	; 转换一个字的字节排列顺序
-----	--------	----------------

REV16	r0, r0	, 分别转换一个字的 2 个半字的字节排列顺序
REVSH	r0, r0	, 转换一个有符号半字的字节排列顺序

RFE 从异常中返回指令

RFE<amode> Rn! ARMv6

如果 LDM 指令允许寄存器列表为 {pc, cpsr}, 则这条 RFE 指令的执行效果相当于指令: LDM<amode> Rn{!}, {pc, cpsr} 的执行效果。请参看 LDM 指令。

ROR Thumb 的循环右移指令(请参看 ARM 的 MOV 指令)

ROR Ld, Ls THUMBv1

Action 对 cpsr 的影响
Ld = Ld ROR Ls[7,0] 更新

注释:

cpsr 更新为: N = <Negative>, Z = <Zero>, C = <shift_C> (见表 A.3)。

RSB 2 个 32 位整型数的反向减法指令

- ① RSB<cond>{S} Rd, Rn, #<rotated_immed> ARMv1
② RSB<cond>{S} Rd, Rn, Rm{, <shift>} ARMv1

Action 对 cpsr 的影响
① Rd = <rotated_immed> - Rn 如果指定了后缀 S, 则更新
② Rd = <shifted_Rm> - Rn 如果指定了后缀 S, 则更新

注释:

- 如果操作要更新 cpsr 并且 Rd 不是 pc, 则 N = <Negative>, Z = <Zero>, C = <NoUnsignedOverflow>, V = <SignedOverflow>。因为减法 $x - y$ 的执行等价于加法 $x + \sim y + 1$, 所以进位位要这样设置。如果加法 $x + \sim y + 1$ 溢出, 则进位位等于 1。这发生在 $x \geq y$ 时, 即 $x - y$ 不会溢出时。
- 如果 Rd 是 pc, 则指令的执行将会影响到下一条指令的执行地址。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 且 cpsr 的值被设置为

spsr 的值。

- 如果 Rn 或者 Rm 是 pc, 则所使用的值为当前指令的地址加上 8 字节。

例如:

```
RSB    r0, r0, #0           ;r0 = ~r0
RSB    r0, r1, r1, LSL #3   ;r0 = 7 * r1
```

RSC

2 个 32 位整型数的带借位的反向减法指令

- ① RSC<cond>{S} Rd, Rn, #<rotated_immed> ARMv1
- ② RSC<cond>{S} Rd, Rn, Rm {, <shift>} ARMv1

Action

对 cpsr 的影响

- ① $Rd = \langle \text{rotated_immed} \rangle - Rn - (\sim C)$ 如果指定了 S, 则更新
- ② $Rd = \langle \text{shifted_Rm} \rangle - Rn - (\sim C)$ 如果指定了 S, 则更新

注释:

- 如果操作要更新 cpsr 且 Rd 不是 pc, 则 N = <Negative>, Z = <Zero>, C = <NoUnsignedOverflow>, V = <SignedOverflow>。因为减法 $x - y - \sim C$ 的执行等价于加法 $x + \sim y + C$, 所以进位位要这样设置。如果加法 $x + \sim y + C$ 溢出, 则进位位等于 1。这发生在 $x - y - \sim C$ 不会溢出时。
- 如果 Rd 是 pc, 则指令的执行将会影响到下一条指令的执行地址。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr, 且 cpsr 的值被设置为 spsr 的值。
- 如果 Rn 或者 Rm 是 pc, 则所使用的值为当前指令的地址加上 8 字节。

如下的代码是对一个 64 位长整型数求相反数, r0 是其低 32 位, r1 是其高 32 位。

```
RSBS   r0, r0, #0           ;r0 = -r0    C = NOT(borrow)
RSC    r1, r1, #0           ;r1 = ~r1 - borrow
```

SADD

分块并行加减运算指令

- ① {S|U} ADD16<cond> Rd, Rn, Rm ARMv6
- ② {S|U} ADDSUBX<cond> Rd, Rn, Rm ARMv6
- ③ {S|U} SUBADDX<cond> Rd, Rn, Rm ARMv6
- ④ {S|U} SUB16<cond> Rd, Rn, Rm ARMv6

- | | | |
|-------------------|------------|-------|
| ⑤ {S U}ADD8<cond> | Rd, Rn, Rm | ARMv6 |
| ⑥ {S U}SUB8<cond> | Rd, Rn, Rm | ARMv6 |

Action

对 cpsr 的影响

- ① $Rd[31:16] = Rn[31:16] + Rm[31:16]$; $GE3 = GE2 \leftarrow cmn(Rn[31:16], Rm[31:16])$
 $Rd[15:00] = Rn[15:00] + Rm[15:00]$ $GE1 = GE0 = cmn(Rn[15:00], Rm[15:00])$
- ② $Rd[31:16] \leftarrow Rn[31:16] + Rm[15:00]$; $GE3 = GE2 \leftarrow cmn(Rn[31:16], Rm[15:00])$
 $Rd[15:00] = Rn[15:00] - Rm[31:16]$ $GE1 = GE0 = (Rn[15:00] \geq Rm[31:16])$
- ③ $Rd[31:16] = Rn[31:16] \ll Rm[15:00]$; $GE3 = GE2 \leftarrow (Rn[31:16] \geq Rm[15:00])$
 $Rd[15:00] = Rn[15:00] + Rm[31:16]$ $GE1 = GE0 = cmn(Rn[15:00], Rm[31:16])$
- ④ $Rd[31:16] = Rn[31:16] - Rm[31:16]$; $GE3 = GE2 = (Rn[31:16] \geq Rm[31:16])$
 $Rd[15:00] = Rn[15:00] - Rm[15:00]$ $GE1 = GE0 = (Rn[15:00] \geq Rm[15:00])$
- ⑤ $Rd[31:24] = Rn[31:24] + Rm[31:24]$; $GE3 = cmm(Rn[31:24], Rm[31:24])$
 $Rd[23:16] = Rn[23:16] + Rm[23:16]$; $GE2 = cmm(Rn[23:16], Rm[23:16])$
 $Rd[15:08] = Rn[15:08] + Rm[15:08]$; $GE1 = cmm(Rn[15:08], Rm[15:08])$
 $Rd[07:00] = Rn[07:00] + Rm[07:00]$ $GE0 = cmm(Rn[07:00], Rm[07:00])$
- ⑥ $Rd[31:24] = Rn[31:24] - Rm[31:24]$; $GE3 = (Rn[31:24] \geq Rm[31:24])$
 $Rd[23:16] = Rn[23:16] - Rm[23:16]$; $GE2 = (Rn[23:16] \geq Rm[23:16])$
 $Rd[15:08] = Rn[15:08] - Rm[15:08]$; $GE1 = (Rn[15:08] \geq Rm[15:08])$
 $Rd[07:00] = Rn[07:00] - Rm[07:00]$ $GE0 = (Rn[07:00] \geq Rm[07:00])$

注释:

- 如果指定了后缀 S, 则所有的比较都是有符号的。函数 $cmn(x, y)$ 返回 $x \geq -y$ 或 $(x + y \geq 0)$ 的值。
- 如果指定了前缀 U, 则所有的比较都是无符号的。函数 $cmn(x, y)$ 返回 $x \geq (\text{unsigned})(-y)$ 的值, 也就是 $x + y$ 是否会产生一个进位。
- Rd, Rn 和 Rm 都一定不能是 pc。
- X 操作对于打包的复数非常有用。下面的例子假定[15:00]位是复数的实部, [31:16]是虚部。

例如:

SADD16	r0, r1, r2	;有符号的 16 位 SIMD 加法
SADDSUBX	r0, r1, r2	; $r0 = r1 + i * r2$
SSUBADDX	r0, r1, r2	; $r0 = r1 - i * r2$

SBC

带借位的减法指令

- ① SBC<cond>{S} Rd, Rn, #<rotated_immed> ARMv1

ARM 嵌入式系统开发

② SBC<cond>{S}	Rd, Rn, Rm {, <shift>}	ARMv1
③ SBC	Ld, Lm	THUMBv1

Action	对 cpsr 的影响
① $Rd = Rn - \langle \text{rotated_immed} \rangle - (\sim C)$	如果指定了后缀 S, 则更新
② $Rd = Rn - \langle \text{shifted_Rm} \rangle - (\sim C)$	如果指定了后缀 S, 则更新
③ $Ld = Ld - Lm - (\sim C)$	更新(见下面的注释)

注释:

- 若操作要更新 cpsr 且 Rd 不是 pc, 则 $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{NoUnsignedOverflow} \rangle$, $V = \langle \text{SignedOverflow} \rangle$ 。因为减法 $x - y - \sim C$ 的执行情况和加法 $x + y + \sim C$ 相同, 所以进位位要这样设置。如果 $x + y + \sim C$ 溢出, 则进位位为 1。这发生在 $x - y - \sim C$ 不会溢出时。
- 如果 Rd 是 pc, 则指令的计算结果将会影响到下一条指令的执行地址。如果操作要更新 cpsr, 则处理器模式一定要有一个 spsr。这样 cpsr 的值被设置成 spsr 的值。
- 如果 Rn 或者 Rm 是 pc, 则使用的值为当前的指令地址加上 8 字节。

如下代码执行了 2 个 64 位数的减法运算:

```
SUBS  r0, r0, r2      ; 低位字的减法, C = NOT(borrow)
SBC   r1, r1, r3      ; 高位字的减法和借位
```

580

SEL 根据 GE 标志从 2 个源操作数中进行选择的指令

SEL<cond> Rd, Rn, Rm ARMv6

Action	
$Rd[31:24] = GE3 ?$	$Rn[31:24] ; Rm[31:24];$
$Rd[23:16] = GE2 ?$	$Rn[23:16] ; Rm[23:16];$
$Rd[15:08] = GE1 ?$	$Rn[15:08] ; Rm[15:08];$
$Rd[07:00] = GE0 ?$	$Rn[07:00] ; Rm[07:00];$

注释:

- Rd, Rn 和 Rm 一定不能是 pc。
- 见 SADD 指令中的 cpsr 的 GE 位的设置情况。

SETEND 为数据存取设置字节排列顺序指令

- | | | |
|----------|----|---------------|
| ① SETEND | BE | ARMv6/THUMBv3 |
| ② SETEND | LE | ARMv6/THUMBv3 |

Action

- ① cpsr 寄存器中 E=1, 所以数据按大端顺序进行存取。
- ② cpsr 寄存器中 E=0, 所以数据按小端顺序进行存取。

注释:

ARMv6 使用字节排列顺序不变的字节排列模式。这就意味着字节的装载和存储都不会受字节排列顺序的配置而变化。对于小端数据存取顺序的字装载操作, 最低地址的字节是最低有效字节; 对于大端数据存取顺序的字装载操作, 最低地址的字节是最高有效字节。

SHADD 并行的除以 2 加减运算指令

- | | | |
|-----------------------|------------|-------|
| ① {S U}HADD16<cond> | Rd, Rn, Rm | ARMv6 |
| ② {S U}HADDSUBX<cond> | Rd, Rn, Rm | ARMv6 |
| ③ {S U}HSUBADDX<cond> | Rd, Rn, Rm | ARMv6 |
| ④ {S U}HSUB16<cond> | Rd, Rn, Rm | ARMv6 |
| ⑤ {S U}HADD8<cond> | Rd, Rn, Rm | ARMv6 |
| ⑥ {S U}HSUB8<cond> | Rd, Rn, Rm | ARMv6 |

Action

- ① $Rd[31:16] = (Rn[31:16] + Rm[31:16]) \gg 1;$
 $Rd[15:00] = (Rn[15:00] + Rm[15:00]) \gg 1$
- ② $Rd[31:16] = (Rn[31:16] + Rm[15:00]) \gg 1;$
 $Rd[15:00] = (Rn[15:00] - Rm[31:16]) \gg 1$
- ③ $Rd[31:16] = (Rn[31:16] - Rm[15:00]) \gg 1;$
 $Rd[15:00] = (Rn[15:00] + Rm[31:16]) \gg 1$
- ④ $Rd[31:16] = (Rn[31:16] - Rm[31:16]) \gg 1;$
 $Rd[15:00] = (Rn[15:00] - Rm[15:00]) \gg 1$
- ⑤ $Rd[31:24] = (Rn[31:24] + Rm[31:24]) \gg 1;$
 $Rd[23:16] = (Rn[23:16] + Rm[23:16]) \gg 1;$
 $Rd[15:08] = (Rn[15:08] + Rm[15:08]) \gg 1;$
 $Rd[07:00] = (Rn[07:00] + Rm[07:00]) \gg 1;$

- ⑥ $Rd[31:24] = (Rn[31:24] - Rm[31:24]) \gg 1;$
 $Rd[23:16] = (Rn[23:16] - Rm[23:16]) \gg 1;$
 $Rd[15:08] = (Rn[15:08] - Rm[15:08]) \gg 1;$
 $Rd[07:00] = (Rn[07:00] - Rm[07:00]) \gg 1$

注释:

- 如果指定了前缀 S, 则所有的操作都是有符号的, 而且在作加法运算之前, 所有的值都要作有符号扩充。
- 如果指定了前缀 U, 则所有的操作都是无符号的, 而且在作加法运算之前, 所有的值都要作 0 扩充。
- Rd, Rn 和 Rm 一定不能是 pc。
- 这些运算都是并行进行的, 所以都不能溢出。这对于 DSP 处理归一化信号是非常有用的。

SMLA

有符号的乘累加运算指令

SMLS

- | | | |
|----------------------|--------------------|--------|
| ① SMLA<x><y><cond> | Rd, Rm, Rs, Rn | ARMv5E |
| ② SMLAW<y><cond> | Rd, Rm, Rs, Rn | ARMv5E |
| ③ SMLAD{X}<cond> | Rd, Rm, Rs, Rn | ARMv6 |
| ④ SMLSd{X}<cond> | Rd, Rm, Rs, Rn | ARMv6 |
| ⑤ {U S}MLAL<cond>{S} | RdLo, RdHi, Rm, Rs | ARMv3M |
| ⑥ SMLAL<x><y><cond> | RdLo, RdHi, Rm, Rs | ARMv5E |
| ⑦ SMLALD{X}<cond> | RdLo, RdHi, Rm, Rs | ARMv6 |
| ⑧ SMLSd{X}<cond> | RdLo, RdHi, Rm, Rs | ARMv6 |

Action

- | | |
|--------------|--|
| ① Rd | $= Rn + (Rm.<x> * Rs.<y>)$ |
| ② Rd | $= Rn + (((signed)Rm * Rs.<y>) \gg 16)$ |
| ③ Rd | $= Rn + Rm.B * \langle rotated_Rs \rangle.B + Rm.T * \langle rotated_Rs \rangle.T$ |
| ④ Rd | $= Rn + Rm.B * \langle rotated_Rs \rangle.B - Rm.T * \langle rotated_Rs \rangle.T$ |
| ⑤ RdHi; RdLo | $= RdHi; RdLo + (Rm * Rs)$ |
| ⑥ RdHi; RdLo | $= RdHi; RdLo + (Rm.<x> * Rm.<y>)$ |
| ⑦ RdHi; RdLo | $= RdHi; RdLo + Rm.B * \langle rotated_Rs \rangle.B + Rm.T * \langle rotated_Rs \rangle.T$ |

⑧ $RdHi; RdLo = RdHi; RdLo + Rm.B * \langle rotated_Rs \rangle.B - Rm.T * \langle rotated_Rs \rangle.T$

注释:

- $\langle x \rangle$ 和 $\langle y \rangle$ 可以是 B 或者 T。
- Rm.B 是 (sign-extend)Rm[15:00] 的缩写形式, Rm 的低 16 位。
- Rm.T 是 (sign-extend)Rm[31:16] 的缩写形式, Rm 的高 16 位。
- 如果没有指定后缀 X, 则 $\langle rotated_Rs \rangle$ 表示 Rs; 若指定了后缀 X, 则表示 Rs ROR 16。
- RdHi 和 RdLo 必须是不同的寄存器。对于第 5 条指令格式, Rm 一定要与 RdHi 和 RdLo 是不同的寄存器。
- 第 1~4 条指令格式更新 cpsr 的 Q 标志位为: $Q = Q | \langle SignedOverflow \rangle$ 。
- 对于第 5 条指令格式, 如果指定了前缀 U, 则是无符号乘法; 若指定了前缀 S, 则是有符号乘法。
- 对于第 5 条指令格式, 如果指定了后缀 S, 则 cpsr 更新为: $N = RdHi[31], Z = (RdHi == 0 \& \& RdLo == 0)$; C 和 V 都是不可预知的。因为经常会产生多余的指令周期而影响到其它指令的执行, 所以应避免使用指令 {U|S}MLALS。
- 因为 Rs 的值可能会导致运算的提前中止执行, 所以 Rs 的值应尽量小或者使用常数。
- 带后缀 X 的指令和乘减指令 (multiply subtract) 对于打包的复数非常有用。下面的例子假设 bits[15:00] 是复数的实部, bits[31:16] 是虚部。

例如:

SMLABB	r0, r1, r2, r0	;r0 += (short)r1 * (short)r2
SMLABT	r0, r1, r2, r0	;r0 += (short)r1 * ((signed)r2 >> 16)
SMLANB	r0, r1, r2, r0	;r0 += (r1 * (short)r2) >> 16
SMLAL	r0, r1, r2, r3	;acc += r2 * r3, acc 是一个 64 位数 bits[r1:r0]
SMLALTB	r0, r1, r2, r3	;acc += ((signed)r2 >> 16) * ((short)r3)
SMLSD	r0, r1, r2, r0	;r0 += real(r1 * r2) 复数运算
SMLADX	r0, r1, r2, r0	;r0 += imag(r1 * r2) 复数运算

SMMUL
SMMLA
SMMLS

有符号的最高有效字乘法(signed most significant word multiply)指令

- | | | |
|------------------|----------------|-------|
| ① SMMUL{R}<cond> | Rd, Rm, Rs | ARMv6 |
| ② SMMLA{R}<cond> | Rd, Rm, Rs, Rn | ARMv6 |
| ③ SMMLS{R}<cond> | Rd, Rm, Rs, Rn | ARMv6 |

Action

- | | |
|---|--|
| ① | $Rd = ((\text{signed})Rm * (\text{signed})Rs + \text{round}) \gg 32$ |
| ② | $Rd = ((Rn \ll 32) + (\text{signed})Rm * (\text{signed})Rs + \text{round}) \gg 32$ |
| ③ | $Rd = ((Rn \ll 32) - (\text{signed})Rm * (\text{signed})Rs + \text{round}) \gg 32$ |

注释:

- 如果指定了后缀 R, 则 $\text{round} = 2^{31}$; 否则 $\text{round} = 0$ 。
- Rd, Rm, Rs 和 Rn 一定不能是 pc。
- 可能会因为 Rs 的值而导致指令执行的提前结束。
- 对于 32 位的 DSP 运算, 这些操作比使用高结果寄存器(high result register)的 SMLAL 指令更有优势: 常常能够使用比 SMLAL 更少的周期; 也可以实现舍入(implement rounding)、乘减, 而且不需要低 32 位的结果的临时存储寄存器(草稿板寄存器)。

例如:

SMMULR r0, r1, r2 ;r0 = r1 * r2/2 使用 Q31 算法

SMUL
SMUA
SMUS

有符号数的乘法指令

- | | | |
|----------------------|--------------------|--------|
| ① SMUL<x><y><cond> | Rd, Rm, Rs | ARMv5E |
| ② SMULW<y><cond> | Rd, Rm, Rs | ARMv5E |
| ③ SMUAD{x}<cond> | Rd, Rm, Rs | ARMv6 |
| ④ SMUSD{x}<cond> | Rd, Rm, Rs | ARMv6 |
| ⑤ {U S}MULL<cond>{S} | RdLo, RdHi, Rm, Rs | ARMv3M |

Action

- | | |
|------|------------------------|
| ① Rd | = Rm. <x> * Rs. <y> |
| ② Rd | = (Rm * Rs. <y>) >> 16 |

- ③ $Rd = Rm.B * \langle rotated_Rs \rangle.B + Rm.T * \langle rotated_Rs \rangle.T$
 ④ $Rd = Rm.B * \langle rotated_Rs \rangle.B - Rm.T * \langle rotated_Rs \rangle.T$
 ⑤ $RdHi;RdLo = Rm * Rs$

注释:

- $\langle x \rangle$ 和 $\langle y \rangle$ 可以是 B 或者 T。
- $Rm.B$ 是 (sign-extend) $Rm[15:00]$ 的缩写形式, 是 Rm 的低 16 位。
- $Rm.T$ 是 (sign-extend) $Rm[31:00]$ 的缩写形式, 是 Rm 的高 16 位。
- 如果没有指定后缀 X, 则 $\langle rotated_Rs \rangle$ 代表 Rs ; 否则代表 Rs ROR 16。
- $RdHi$ 和 $RdLo$ 必须是不同的寄存器。对于第 5 条指令格式, Rm 必须是与 $RdHi$ 和 $RdLo$ 不同的寄存器。
- 第 4 条指令格式更新 cpsr 的 Q-flag 为: $Q = Q | \langle SignedOverflow \rangle$ 。
- 对于第 5 条指令格式, 如果指定了前缀 U, 则是无符号乘法; 若指定了前缀 S, 则是有符号乘法。
- 对于第 5 条指令格式, 如果指定了后缀 S, 则 cpsr 更新为: $N = RdHi[31], Z = (RdHi == 0 \& \& RdLo == 0), C$ 和 V 是不可预知的。因为经常会产生更多的指令周期而影响到其它指令的执行, 所以应避免使用指令 $\{U|S\}MULLS$ 。
- 可能会因为 Rs 的值使指令的执行提前中止。所以 Rs 应尽可能地使用比较小的值或者使用常数。
- 后缀 X 和乘减 (multiply subtract versions) 对于打包的复数非常有用。下面的例子假设 $bits[15:00]$ 是复数的实部, $bits[31:16]$ 是虚部。

例如:

```
SMULBB    r0, r1, r2      ;r0 = (short)r1 * (short)r2
SMULBT    r0, r1, r2      ;r0 = (short)r1 * ((signed)r2 >> 16)
SMULNB    r0, r1, r2      ;r0 = (r1 * (short)r2) >> 16
SMULL     r0, r1, r2, r3   ;acc = r2 * r3, acc 是一个 64 位数[r1;r0]
SMUADX    r0, r1, r2      ;r0 = imag(r1 * r2)复数运算
```

SRS

存储返回状态指令

SRS<amode> #<mode>{!}

ARMv6

如果 STM 指令允许寄存器列表 {lr, spsr} 而且允许以不同的模式调用栈指针, 则上面的指令和 STM<amode> sp_<mode>{!}, {lr, spsr} 所要执行的操作一样。参看 STM 指令。

SSAT把一个数饱和至 n 位

- ① {S|U}SAT<cond> Rd, #< n >, Rm {, LSL #<0-31>}
 ② {S|U}SAT<cond> Rd, #< n >, Rm {, ASR #<1-32>}
 ③ {S|U}SAT16<cond> Rd, #< n >, Rm

Action

对 cpsr 的影响

- ① $Rd = \text{sat}(\text{<shifted_Rm>}, n)$; $Q = Q \mid 1$ 如果发生了饱和
 ② $Rd = \text{sat}(\text{<shifted_Rm>}, n)$; $Q = Q \mid 1$ 如果发生了饱和
 ③ $Rd[31:16] = \text{sat}(Rm[31:16], n)$; $Q = Q \mid 1$ 如果发生了饱和
 $Rd[15:00] = \text{sat}(Rm[15:00], n)$

注释:

- 如果指定了前缀 S, 则 $\text{sat}(x, n)$ 饱和一个有符号数 x 到一个 n 位的有符号数, x 的范围是 $-2^{n-1} \leq x < 2^{n-1}$ 。对于 SAT 指令, n 的值为 $1 + \text{<immed5>}$; 对于 SAT16 指令, n 的值为 $1 + \text{<immed4>}$ 。
- 如果指定了前缀 U, 则 $\text{sat}(x, n)$ 饱和一个有符号数 x 到一个 n 位的无符号数, x 的范围是 $0 \leq x < 2^n$ 。对于 SAT 指令, n 的值为 <immed5> ; 对于 SAT16 指令, n 的值为 <immed4> 。
- Rd 和 Rm 一定不能是 pc。

SSUB

有符号数的并行减法指令(见 SADD 指令)

STC

一个或多个 32 位字的协处理器数据写入指令

- ① STC<cond>{L} <copro>, Cd, [Rn {, #<-><immed8>*4}] {!} ARMv2
 ② STC<cond>{L} <copro>, Cd, [Rn], #<-><immed8>*4 ARMv2
 ③ STC<cond>{L} <copro>, Cd, [Rn], <option> ARMv2
 ④ STC2{L} <copro>, Cd, [Rn {, #<-><immed8>*4}] {!} ARMv5
 ⑤ STC2{L} <copro>, Cd, [Rn], #<-><immed8>*4 ARMv5
 ⑥ STC2{L} <copro>, Cd, [Rn], <option> ARMv5

这些指令初始化一个存储器的写, 从给定的协处理器把数据传送到存储器去。<copro>是 p0~p15 的协处理器号。如果系统没有指定的协处理器, 则将会产生一个未定义指令的陷阱。存储器写是通过地址的增加来访问一系列连续的存储器单元。初始地址是通过表 A. 10 的寻址模式指定的。协处理

器控制着传送的字数,最大不超过 16 个字。域{L}和 Cd 都由协处理器解释,ARM 将忽略它们。通常情况下,Cd 指定了要传送的协处理寄存器。域<option>是一个 8 位整型数,由{}括起来,其解释依赖于协处理器。

如果地址不是 4 的倍数,则地址访问是未对齐的。对于未对齐的地址访问的限制和 STM 一样,可以参照 STM。

表 A.10 STC 的寻址模式

寻址模式	地址入口	回写给 Rn 的值
$[Rn\{, \# \{-\} \langle immed \rangle \}]$	$Rn + \{ \{-\} \langle immed \rangle \}$	Rn 是受保护的
$[Rn\{, \# \{-\} \langle immed \rangle \}]!$	$Rn + \{ \{-\} \langle immed \rangle \}$	$Rn + \{ \{-\} \langle immed \rangle \}$
$[Rn], \# \{-\} \langle immed \rangle$	Rn	$Rn + \{-\} \langle immed \rangle$
$[Rn], \langle option \rangle$	Rn	Rn 是受保护的

STM

把一组 32 位寄存器的值写入存储器指令

- ① $STM \langle cond \rangle \langle a\ mode \rangle\ Rn\{!\}, \langle register_list \rangle \{^-\}$ ARMv1
 ② STMIA $Rn!, \langle register_list \rangle$ THUMBv1

这些指令存储一组字到连续的存储器单元中去。<register_list>由{}括起来,指定了一个要被存储的寄存器列表。虽然汇编语言允许以任意的顺序在寄存器列表中排列寄存器,它们的顺序不会被指令记下来,但是由于存储器的访问顺序通常是从小到大的,所以最好在寄存器列表中按照寄存器号增加的顺序排列寄存器。

下面的伪代码演示了正常情况下 STM 的操作过程。在这里使用<register_list>[i]来表示寄存器列表中出现第 i 个位置的寄存器,下标从 0 开始计数。这里假定寄存器列表的寄存器按照寄存器号增加的顺序排列。

```

N = 寄存器列表<register_list>中的寄存器个数
start = 在表 A.11 中的最低访问地址
for (i = 0, i < N; i++)
    memory(start + i * 4, 4) = <register_list>[i];
if (指定 '!') then 根据表 A.11 更新 Rn
  
```

表 A.11 STM 的寻址模式

寻址模式	访问存储器的 最低地址	访问存储器的 最高地址	如果指定了“!”, 则回写给 Rn 的值
{IA EA}	R_n	$R_n + N * 4 - 4$	$R_n + N * 4$
{IB FA}	$R_n + 4$	$R_n + N * 4$	$R_n + N * 4$
{DA ED}	$R_n - N * 4 + 4$	R_n	$R_n - N * 4$
{DB FD}	$R_n - N * 4$	$R_n - 4$	$R_n - N * 4$

这里 $\text{memory}(a, 4)$ 代表根据当前处理器的字节排列顺序, 从地址 a 开始打包的 4 字节。如果地址 a 不等于 4 的倍数, 则地址访问是没有对齐的。因为对于没有对齐的地址访问依赖体系结构的版本、存储器系统和系统协处理器 (CP15) 的配置, 所以应尽量避免不对齐的写入操作。如果外部存储器系统不会因为不对齐的装载操作而异常中止, 则通常会遵照下面的规则:

- 如果内核有一个系统协处理器, 且 CP15: c1: c0: 0 的位 1 (A_bit) 或者位 22 (U_bit) 被置位, 则没对齐的批量存储器写入将会产生一个没对齐的数据异常中断。
- 否则, 操作将会忽略最低的 2 地址位。

表 A.11 列出了由 $\langle \text{amode} \rangle$ 指定的可能的几种寻址模式。如果还指定了后缀 “!”, 则基址寄存器将按照表 A.11 的方式更新; 否则它是受保护的。

注意: 通常最低编号的寄存器数据保存在最低地址的存储器中。

寻址模式的前一半分别代表 IA (过后增加, Increment After)、IB (预先增加, Increment Before)、DA (过后减少, Decrement After)、DB (预先减少, Decrement Before)。在增加模式下, 过后增加从地址 R_n 开始, 预先增加从地址 $R_n + 4$ 开始, 顺序地向前保存寄存器列表的寄存器。在减少模式下, 如果采用向后读取寄存器列表的寄存器, 并且按照存储器地址减少的方式存储数据, 则和增加模式有一样的效果。但是起始地址分别是过后减少地址从 R_n 开始, 预先减少地址从 $R_n - 4$ 开始。

寻址模式的后一半代表栈的类型: 是满栈或是空栈, 是升序栈或是降序栈; FD (满降序, Full Descending)、ED (空降序, Empty Descending)、FA (满升序, Full Ascending)、EA (空升序, Empty Ascending)。对于满栈, R_n 指向栈的最后一个值; 对于空栈, R_n 指向栈的第一个未使用的栈的位置。ARM

的栈通常是满降序。可以根据自己的偏好来选择使用满降序或是空升序，STC 指令同样适合这些寻址模式。

注释：

- 对于 Thumb(格式 2), R_n 和寄存器列表中的寄存器必须是 $r0 \sim r7$ 中的。
- 列表中的寄存器个数 N 必须是非 0 的。
- R_n 不能是 pc 。
- 如果 R_n 出现在寄存器列表中, 而且指定了“!”(回写), 则操作如下:
- 如果 R_n 是列表中最低编号的寄存器, 则保存最初的值; 否则 R_n 的值是不可预知的。
- 如果 pc 出现在寄存器列表中, 则存储的值是在实现时被指定的。
- 如果指定了“-”, 则操作产生了变化, 处理器一定不能处于用户模式或者系统模式。出现在寄存器列表的寄存器参照寄存器的用户模式版本, 而且一定不能指定回写。
- 存储器读取的时序依赖于指令的执行情况。在使用多字节存取指令处理 I/O 时应注意时序问题。如果与时序有关, 则应检查存储器的位置是否在页表内。不要超出页表边界, 而且不要让 pc 在寄存器列表中。

例如：

```
STMLIA    r4!, {r0, r1}    ; *r4 = r0, *(r4+4) = r1, r4 += 8
STMDB     r4!, {r0, r1}    ; *(r4-4) = r1, *(r4-8) = r0, r4 -= 8
STMEQFD   sp!, {r0, lr}    ; if(result zero) then stack r0, lr
STMFID    sp, {sp}^        ; store sp_usr on stack sp_current
```

STR

保存一个值到指定的存储器地址

- ① STR<cond>{ |B} Rd, [Rn {, # { - } <immed12> }] { ! } ARMv1
- ② STR<cond>{ |B} Rd, [Rn, { - } Rm {, <imm_shift> }] { ! } ARMv1
- ③ STR<cond>{ |B } { T } Rd, [Rn], # { - } <immed12> ARMv1
- ④ STR<cond>{ |B } { T } Rd, [Rn], { - } Rm {, <imm_shift> } ARMv1
- ⑤ STR<cond>{ H } Rd, [Rn, {, # { - } <immed8> }] { ! } ARMv4
- ⑥ STR<cond>{ H } Rd, [Rn, { - } Rm] { ! } ARMv4
- ⑦ STR<cond>{ H } Rd, [Rn], # { - } <immed8> ARMv4
- ⑧ STR<cond>{ H } Rd, [Rn], { - } Rm ARMv4
- ⑨ STR<cond>D Rd, [Rn, {, # { - } <immed8> }] { ! } ARMv5E
- ⑩ STR<cond>D Rd, [Rn, { - } Rm] { ! } ARMv5E

ARM 嵌入式系统开发

⑪ STR<cond>D	Rd, [Rn], # {-}<immed8>	ARMv5E
⑫ STR<cond>D	Rd, [Rn], {-}Rm	ARMv5E
⑬ STREX<cond>	Rd, Rm, [Rn]	ARMv6
⑭ STR{ B H}	Ld, [Ln, #<immed5> * <size>]	THUMBv1
⑮ STR{ B H}	Ld, [Ln, Ln]	THUMBv1
⑯ STR	Ld, [sp, #<immed8> * 4]	THUMBv1
⑰ STR<cond><type> Rd, <label>		MACRO

第 1~16 条指令格式存储单一的数据,数据类型由操作码后缀指定,使用前变址或者后变址的寻址模式。表 A.12 和表 A.13 分别列出了不同的寻址模式和数据类型。

表 A.12 STR 寻址模式

寻址模式	入口地址 a	回写给 Rn 的值
[Rn {, # {-}<immed>}]	$Rn + \{-\}\langle immed \rangle$	Rn 受保护的
[Rn {, # {-}<immed>}]!	$Rn + \{-\}\langle immed \rangle$	$Rn + \{-\}\langle immed \rangle$
[Rn, {-}Rm {, <shift>}]	$Rn + \{-\}\langle shifted_Rm \rangle$	Rn 受保护的
[Rn, {-}Rm {, <shift>}]!	$Rn + \{-\}\langle shifted_Rm \rangle$	$Rn + \{-\}\langle shifted_Rm \rangle$
[Rn], # {-}<immed>	Rn	$Rn + \{-\}\langle immed \rangle$
[Rn], {-}Rm {, <shift>}	Rn	$Rn + \{-\}\langle shifted_Rm \rangle$

表 A.13 STR 的数据类型

Store	数据类型	大小/字节	操作
STR	word	4	$memory(a, 4) = Rd$
STRB	unsigned Byte	1	$memory(a, 1) = (char)Rd$
\$STRBT	Byte Translated	1	$memoryT(a, 1) = (char)Rd$
STRD	Double word	8	$memory(a, 4) = Rd$ $memory(a+4, 4) = R(d+1)$

续表 A.13

Store	数据类型	大小/字节	操 作
STREX	word EXclusive	4	if(IsExclusive(a)) { memory(a, 4) = Rd; Rd = 0; } else { Rd = 1; }
STRH	unsigned Halfword	2	memory(a, 2) = (short)Rd
STRT	word Translated	4	memoryT(a, 4) = Rd

在表 A.13 中, $\text{memory}(a, n)$ 从地址 a 开始, 顺序地存储 n 个字节的数据。数据的打包根据当前处理器的字节排列顺序不同而不同。不管当前的处理器模式是什么, 函数 $\text{memoryT}(a, n)$ 都遵从特权级用户模式要求。函数 $\text{IsExclusive}(a)$ 依赖于地址 a , 由 STREX 指令调用。如果地址 a 有共享 TLB 属性, 而且地址 a 被标记为这个处理器的独占访问地址, 则 $\text{IsExclusive}(a)$ 为真。然后清除该处理器的其它独占访问地址和在该系统下地址 a 在其它处理器上的独占访问标志。如果 a 没有共享的 TLB 属性, 而且这个处理器有一个明显的独占访问, 则 $\text{IsExclusive}(a)$ 为真。然后清除所有这样的访问。

如果地址 a 不是 $\langle \text{size} \rangle$ 的倍数, 则存储指令是未对齐的。因为对于未对齐的存储操作, 依赖于系统体系结构的版本、存储器系统和协处理器系统 (CP15) 的配置, 所以应尽量避免未对齐的存储操作。假定外部的存储系统不会因为不对齐读取数据而产生异常中断, 则通常会按照如下的规则执行。在该规则下, A 是协处理器寄存器 CP15:c1:c0:0 的位 1, U 是协处理器寄存器 CP15:c1:c0:0 的位 22, 这是在 ARMv6 下提出的。如果系统没有协处理器, 则 $A = U = 0$ 。

- 如果 $A = 1$, 则除在 $U = 1$ 的字对齐的双字存储操作外, 都会因为不对齐的存储操作而产生一个数据对齐错误的异常中断。
- 如果 $A = 0$ 和 $U = 1$, 则 STR{[T|H|SH]} 支持不对齐的存储操作, STRD 支持字对齐的存储操作。一个非字对齐 (non-word-aligned) 的 STRD 读取操作将会产生一个数据对齐错误的异常中断。
- 如果 $A = 0$ 和 $U = 0$, 则 STR 和 STRT 写给 $\text{memory}(a \& \sim 3, 4)$ 。所有的其它不对齐的操作结果都是不可预知的, 但是不会产生一个对齐错误。

第 17 条指令通过由 $\langle \text{label} \rangle$ 指定的地址产生一个和 pc 相关的存储操作。也就是说,当这条指令被支持,而且 $\langle \text{offset} \rangle = \langle \text{label} \rangle - \text{pc}$ 在范围内,则这条指令被汇编为 $\text{STR}\langle \text{cond} \rangle\langle \text{type} \rangle \text{ Rd}, [\text{pc}, \# \langle \text{offset} \rangle]$ 。

注释:

- 对于双字的存储操作(第 9~12 条),Rd 必须是 r0~r12 的偶数号寄存器。
- 如果寻址模式更新了 Rn,则 Rd 和 Rn 一定要是不同的寄存器。
- 如果 Rd 是 pc,则 $\langle \text{size} \rangle$ 一定要等于 4。存储的值是在运行时才能指定的。
- 如果 Rn 是 pc,则寻址模式一定不能更新 Rn。Rn 的值等于当前指令的地址加上 8 字节。
- Rm 一定不能是 pc。

例如:

STR	r0, [r0]	; *(int *)r0 = r0;
STRH	r0, [r1], #4	; *(short *)r1 = r0; r1 += 4;
STRD	r2, [r1, #-8]!	; r1 -= 8; *(int *)r1 = r2; *(int *) (r1 + 4) = r3
STRB	r0, [r2, #55]	; *(char *) (r2 + 55) = r0;
STREB	r0, [r1], -r2, LSL #8	; *(char *)r1 = r0; r1 -= 256 * r2;

SUB

32 位数的减法指令

- | | | |
|--|---|---------|
| ① SUB $\langle \text{cond} \rangle\{S\}$ | Rd, Rn, # $\langle \text{rotated_immed} \rangle$ | ARMv1 |
| ② SUB $\langle \text{cond} \rangle\{S\}$ | Rd, Rn, Rm {, $\langle \text{shift} \rangle$ } | ARMv1 |
| ③ SUB | Ld, Ln, # $\langle \text{immed3} \rangle$ | THUMBv1 |
| ④ SUB | Ld, # $\langle \text{immed8} \rangle$ | THUMBv1 |
| ⑤ SUB | Ld, Ln, Lm | THUMBv1 |
| ⑥ SUB | sp, # $\langle \text{immed7} \rangle * 4$ | THUMBv1 |

Action

对 cpsr 的影响

- | | |
|---|---------------|
| ① Rd = Rn - $\langle \text{rotated_immed} \rangle$ | 如果指定了后缀 S,则更新 |
| ② Rd = Rn - $\langle \text{shifted_Rm} \rangle$ | 如果指定了后缀 S,则更新 |
| ③ Ld = Ln - $\langle \text{immed3} \rangle$ | 更新(见下面的注释) |
| ④ Ld = Ln - $\langle \text{immed8} \rangle$ | 更新(见下面的注释) |
| ⑤ Ld = Ln - Lm | 更新(见下面的注释) |
| ⑥ sp = sp - $\langle \text{immed7} \rangle * 4$ | 受保护的 |

注释:

- 如果操作要更新 cpsr, 并且 Rd 不是 pc, 则 $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{NoUnsignedOverflow} \rangle$, $V = \langle \text{SignedOverflow} \rangle$ 。因为减法 $x - y$ 的执行相当于加法 $x + \sim y + 1$, 所以进位位按照这种方式来设置。如果 $x + \sim y + 1$ 溢出, 则进位位是 1。这发生在 $x \geq y$, 即 $x - y$ 不溢出时。
- 如果 Rd 是 pc, 则这条指令将会影响到下一条指令的地址。如果处理器更新 cpsr, 则处理器模式一定要有一个 spsr, 在这种情况下, cpsr 的值按照 spsr 来设置。
- 如果 Rn 或者 Rm 是 pc, 则所使用的值是这条指令地址加 8 字节。

例如:

```

SUBS    r0, r0, #1           ; r0 -= 1, 设置 flags
SUB     r0, r1, r1, LSL #2   ; r0 = -3 * r1
SUBS    pc, lr, #4           ; jump to lr - 4, set cpsr = spsr

```

SWI 软中断指令

- ① SWI<cond> <immed24> ARMv1
 ② SWI <immed8> THUMBv1

SWI 指令将会导致 ARM 进入超级用户模式, 执行转移到 SWI 向量。返回地址和 cpsr 分别保存在 lr_svc 和 spsr_svc 中。处理器切换到 ARM 状态, IRQ 中断都被禁止。SWI 向量地址是 0x00000008, 但是如果被设置成高向量模式(high vectors), 则向量地址是 0xFFFF0008。

立即操作数将被 ARM 忽略。它通常被 SWI 异常句柄当成一个参数, 用来决定采用哪个函数执行中断。

例如:

```
SWI      0x123456 ; ARM 工具用来实现不完全主机模式(Semi-Hosting)
```

SWP 寄存器和存储器中的单字交换, 不能中断

- ① SWP<cond> Rd, Rm, [Rn] ARMv2a
 ② SWP<cond>B Rd, Rm, [Rn] ARMv2a

Action

- ① temp=memory(Rn, 4); memory(Rn, 4)=Rm; Rd=temp;
 ② temp=(zero extend)memory(Rn, 1); memory(Rn, 1)=(char)Rm;
 Rd=temp;

注释:

- 操作具有原子性,它们不能在执行过程中被打断。
- Rd, Rm 和 Rn 都不能是 pc。
- Rn 和 Rm 必须是不同的寄存器。Rn 和 Rd 必须是不同的寄存器。
- Rn 必须按照存储器大小来对齐。
- 如果在装载时发生一个数据的异常,则存储不会发生;如果在存储时发生一个数据的异常,则 Rd 不会被写入。

在 ARMv5 或者以下的体系结构中,可以使用 SWP 指令来执行 8 位或 32 位的信号量(semaphores)操作。对于 ARMv6,优先选择使用 LDREX 和 STREX。作为一个例子,假定由 r1 指向的一字节信号量寄存器可以有值 0xFF(表示被声明),或者 0x00(表示释放)。接下来的例子是声明锁(claims the lock)。如果一个锁已经被声明(claimed),则代码循环,直到一个中断或任务切换来释放(free)。

```

                MOV    r0, # 0xFF          ;0xFF:声明锁需要的值
Loop           SWPB   r0, r0, [r1]        ;尝试声明锁
                CMP    r0, # 0xFF          ;检查是否已经被声明
                BEQ     loop                ;如果是,等待它被释放

```

SXT

字或半字的提取指令或者字和半字的带累加的提取指令

SXTA

- | | | |
|---------------------|--------------------------------|---------|
| ① {S U}XTB16<cond> | Rd, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ② {S U}XTB<cond> | Rd, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ③ {S U}XTH<cond> | Rd, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ④ {S U}XTAB16<cond> | Rd, Rn, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ⑤ {S U}XTAB<cond> | Rd, Rn, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ⑥ {S U}XTAH<cond> | Rd, Rn, Rm {, ROR # 8 * <rot>} | ARMv6 |
| ⑦ {S U}XTB | Ld, Lm | THUMBv3 |
| ⑧ {S U}XTH | Ld, Lm | THUMBv3 |

Action

- ① $Rd[31:16] = \text{extend}(\langle \text{shifted_Rm} \rangle[23:16]);$
 $Rd[15:00] = \text{extend}(\langle \text{shifted_Rm} \rangle[07:00])$
- ② $Rd = \text{extend}(\langle \text{shifted_Rm} \rangle[07:00])$
- ③ $Rd = \text{extend}(\langle \text{shifted_Rm} \rangle[15:00])$

- ④ $Rd[31;16] = Rn[31;16] + \text{extend}(\langle\text{shifted_Rm}\rangle[23;16]);$
 $Rd[15;00] = Rn[15;00] + \text{extend}(\langle\text{shifted_Rm}\rangle[07;00])$
 ⑤ $Rd = Rn + \text{extend}(\langle\text{shifted_Rm}\rangle[07;00])$
 ⑥ $Rd = Rn + \text{extend}(\langle\text{shifted_Rm}\rangle[15;00])$
 ⑦ $Ld = \text{extend}(Lm[07;00])$
 ⑧ $Ld = \text{extend}(Lm[15;00])$

注释:

- 如果指定前缀 S, 则 $\text{extend}(x)$ 有符号的扩展 x 。
- 如果指定前缀 U, 则 $\text{extend}(x)$ 0 扩展 x 。
- Rd 和 Rm 都不能是 pc 。
- $\langle\text{rot}\rangle$ 是一个 0~3 的立即数。

TEQ

测试 2 个 32 位数是否相等

- ① $\text{TEQ}\langle\text{cond}\rangle \quad Rn, \# \langle\text{rotated_immed}\rangle \quad \text{ARMv1}$
 ② $\text{TEQ}\langle\text{cond}\rangle \quad Rn, Rm \{, \langle\text{shift}\rangle\} \quad \text{ARMv1}$

Action

- ① 根据 $(Rn \wedge \langle\text{rotated_Rm}\rangle)$ 的结果设置 $cpsr$ 。
 ② 根据 $(Rn \wedge \langle\text{shifted_Rm}\rangle)$ 的结果设置 $cpsr$ 。

注释:

- $cpsr$ 更新为: $N = \langle\text{Neagtive}\rangle, Z = \langle\text{Zero}\rangle, C = \langle\text{shifter_C}\rangle$ (见表 A.3)。
- 如果 Rn 或者 Rm 是 pc , 则所用到的值等于当前指令的地址加上 8 字节。
- 如果想检查 2 个数是否相等, 而不想影响到进位位, 则可以使用该指令来代替 CMP 指令。

例如:

```
TEQ    r0, #1           ;test to see if r0 == 1
```

TST

测试位

- ① $\text{TST}\langle\text{cond}\rangle \quad Rn, \# \langle\text{rotated_immed}\rangle \quad \text{ARMv1}$
 ② $\text{TST}\langle\text{cond}\rangle \quad Rn, Rm \{, \langle\text{shift}\rangle\} \quad \text{ARMv1}$
 ③ $\text{TST} \quad Ln, Lm \quad \text{THUMBv1}$

Action

- ① 根据($Rn \& \langle \text{rotated_immed} \rangle$)的值设置 cpsr。
- ② 根据($Rn \& \langle \text{shifted_Rm} \rangle$)的值设置 cpsr。
- ③ 根据($Ln \& Lm$)的值设置 cpsr。

注释:

- cpsr 更新为: $N = \langle \text{Negative} \rangle, Z = \langle \text{Zero} \rangle, C = \langle \text{shifter_C} \rangle$ (见表 A.3)。
- 如果 Rn 或者 Rm 是 pc, 则所使用的值是当前指令地址加上 8 字节。
- 使用该指令来测试所选择的位是否都是 0。

例如:

TST $r0, \#0xFF$, 测试 $r0$ 的低 8 位是否都是 0

UADD 无符号数的分块并行加法指令(见 SADD)

UHADD 无符号数的除以 2 加减运算指令(见 SHADD)

UHSUB

UMAAL 无符号乘累加指令

UMAAL<cond> $RdLo, RdHi, Rm, Rs$ ARMv6

Action

$RdHi, RdLo = (\text{unsigned})Rm * Rs + (\text{unsigned})RdLo + (\text{unsigned})RdHi$

注释:

- $RdHi$ 和 $RdLo$ 必须是不同的寄存器。
- $RdHi, RdLo, Rm, Rs$ 都不能是 pc。
- 因为 $(2^{32}-1)(2^{32}-1) + (2^{32}-1) + (2^{32}-1) = (2^{64}-1)$, 所以操作是不会溢出的。可以在公钥密码系统中使用它来合成多个字的乘法运算。

UMLAL 无符号的 long 类型乘法和乘累加指令(见 SMLAL 和 SMULL)

UMULL

UQADD 饱和的无符号加减运算指令(见 QADD)
UQSUB

USAD 无符号数差分的绝对值求和指令

① USAD8<cond> Rd, Rm, Rs ARMv6

② USADA8<cond> Rd, Rm, Rs, Rn ARMv6

Action

① $Rd = \text{abs}(Rm[31:24] - Rs[31:24]) + \text{abs}(Rm[23:16] - Rs[23:16])$
 $+ \text{abs}(Rm[15:08] - Rs[15:08])$
 $+ \text{abs}(Rm[07:00] - Rs[07:00])$

② $Rd = Rn + \text{abs}(Rm[31:24] - Rs[31:24]) + \text{abs}(Rm[23:16]$
 $- Rs[23:16]) + \text{abs}(Rm[15:08] - Rs[15:08]) + \text{abs}(Rm[07:00] - Rs$
 $[07:00])$

注释:

- $\text{abs}(x)$ 返回 x 的绝对值。Rm 和 Rs 被当作无符号数处理。
- Rd, Rm 和 Rs 不能是 pc。
- 因为在视频编码中提供了一种衡量标准来衡量 2 幅图片的相似程度, 所以这种差分的绝对值求和运算在视频编码中是非常普遍的。

USAT 无符号数的饱和指令(见 SSAT 指令)

USUB 无符号数的分块并行减法指令(见 SADD 指令)

UXT 无符号数的提取和带累加的无符号数提取指令(见 SXT 指令)

UXTA

A.4 ARM 汇编速查

本节概要地列出由 ARM 汇编器——armasm 提供的更有用的命令行和表达式。每一个汇编行都以如下的格式给出:

```

{<label>}      {<instruction>}           ;注释
{<symbol>}     <directive>              ;注释
{<arg_0>}      <macro> {<arg_1>} {, <arg_2>} ...{, <arg_n>} ;注释

```

注释:

- *<instruction>* 要汇编的处理器所支持的所有的 ARM 或 Thumb 指令。见 A.3 节。
- *<label>* 一个用于存储指令地址的符号名。
- *<directive>* 一个 ARM 汇编保留字。见 A.4.4 小节。
- *<symbol>* 被 *<directive>* 所使用的标号名。
- *<macro>* 使用 MACRO 保留字的一个新的保留字定义名。
- *<arg_k>* 第 *k* 个宏参数。

必须在任何 ARM 或 Thumb 指令出现之前,用一个 AREA 保留字定义一个范围。所有汇编文件必须用 END 保留字表示结束。下面的例子演示了一个简单的汇编文件,它定义了一个返回 2 个输入参数和的加法函数。

```

AREA      maths_routines, CODE, READONLY
EXPORT    add           ; 给出一个加法外部联结的标号

add       ADD          r0, r0, r1      ; 输入参数相加
          MOV          pc, lr         ; 从子程序返回

END

```

A.4.1 ARM 汇编变量

ARM 汇编器支持 3 种类型的汇编变量(见表 A.14)。变量名是区分大小写的,而且必须在使用前用保留字 GBLx 或 LCLx 进行声明。

表 A.14 ARM 汇编变量类型

变量类型	全局声明	声明一个宏的 局部变量	设置变量值	示例值
无符号 32 位整型数	GBLA	LCLA	SETA	15, 0xab
ASCII 字符串	GBLS	LCLS	SETS	"", "ADD"
逻辑变量	GBLL	LCLL	SETL	{TRUE}, {FALSE}

可以在表达式中使用变量(见 A.4.2 小节),也可以使用 \$ 操作符在汇编时代替变量。通常, \$ name. 被替换为在该行汇编之前 name 变量的值。如果 name 后面没有跟着字母、数字或下划线,则可以省略最后的句点。使用 \$\$ 来表示符号 \$。算术变量被替换为一个扩展为 8 位十六进制阿拉伯数字的字符串。逻辑变量由 T 或 F 替换。

下面的示例代码演示了如何声明和替换各种类型的变量:

```

;算术变量
GBLA      count      ;声明一个整型的变量 count
count    SETA      1      ;设置 count = 1
        WHILE      count<15
            BL      test $count ;call test00000001,test00000002...
count    SETA      count + 1 ;...test0000000E
        WEND

;字符串变量
GBLS      cc          ;声明一个字符串变量 cc
cc        SETS      "NE" ;设置 cc = "NE"
        ADD $cc      r0,r0,r0 ;汇编成 ADDNE r0,r0,r0
        STR $cc.B     r0,[r1] ;汇编成 STRNEB r0,[r1]

;逻辑变量
GBLL      debug       ;声明一个逻辑变量 debug
debug    SETL      {TRUE} ;设置 debug = {TRUE}
        IF          debug ;if debug is TRUE then
            BL      print_debug ;print out some debug information
        ENDIF

```

A.4.2 ARM 汇编标注

一个标注的定义必须在一行的顶格书写。汇编器把所有不顶格的语句当成指令、保留字或宏。汇编器把格式为<N><name>的标注当成本地标注,其中<N>是一个 0~99 的整数,<name>是一个文本中可选的名字。本地标注由 ROUT 关键字限定范围。通过%{|F|B} {|A|T}<N>{<name>}来引用一个本地标注。额外的前缀字母告诉汇编器如何去查找这个标注:

- 如果指定了前缀 F,则汇编器向前查找;如果指定了前缀 B,则汇编器向后查找。否则汇编器先向后再向前查找。

ARM 嵌入式系统开发

- 如果指定了前缀 T, 则汇编器只查找当前的宏; 如果指定了前缀 A, 则汇编器查找所有级别的宏。否则汇编器先查找当前的宏, 再查找更高级别的嵌套层的宏。

A.4.3 ARM 汇编表达式

ARM 汇编器可以在汇编时分析出许多数字的、字符串的和逻辑的表达式。表 A.15 列出了表达式可以使用的一些一目和二目的操作运算。括号可以用来改变运算顺序。

表 A.15 ARM 一目和二目汇编操作码

表达式	结 果	例 子
A+B, A-B	A 加 B, A 减 B	1-2 = 0xffffffff
A*B, A/B	A 乘以 B, A 除以 B	2*3 = 6, 7/3 = 2
A:MOD:B	A 以 B 为模	7:MOD:3 = 1
:CHR:A	A 转化成 ASCII 码	:CHR:32 = " "
'X'	X 的 ASCII 码值	'a' = 0x61
:STR:A, :STR:L	A 或者 L 转化为一个字符串	:STR:32 = "00000020" :STR:{TRUE} = "T"
A<<B, A:SHL:B	A 左移 B 位	1<<3 = 8
A>>B, A:SHR:B	A 右移 B 位(逻辑右移)	0x80000000 >> 4 = 0x08000000
A:ROR:B A:ROL:B	A 循环右移/左移 B 位	1:ROR:1=0x80000000 0x80000000:ROL:1=1
A=B, A>B, A>=B, A< B, A<=B, A/=B, A<>B	算术或者字符串变量比较(/=和<>都是不等的意思)	(1=2) = {FALSE}, (1<2) = {TRUE}, ("a"="c") = {FALSE}, ("a"<"c") = {TRUE}
A:AND:B, A:OR:B, A:EOR:B, :NOT:A	A 和 B 的按位“与”、“或”、“异或”; A 的按位取反	1:AND:3 = 1 1:OR:3 = 3 :NOT:0 = 0xffffffff
:LEN:S	字符串 S 的长度	:LEN,"ABC" = 3
S:LEFT:B, S:RIGHT:B	S 的最左或最右 B 位字母	"ABC":LEFT:2 = "AB", "ABC":RIGHT:2 = "BC"

续表 A. 15

表达式	结 果	例 子
S;CC;T	S 和 T 串联成一个字符串	"AB";CC; "C" = "ABC"
L;LAND;M, L;LOR;M, L;LEOR;M	L 和 M 的逻辑“与”、“或”、“异或”	{TRUE}; LAND; {FALSE} = {FALSE}
;DEF;X	如果变量 X 是定义过的,则返回真	
;BASE;A	见 MAP 关键字	
;INDEX;A		

在表 A. 15 中,A 和 B 代表任意的整型数;S 和 T 代表字符串;L 和 M 代表逻辑值。在很多表达式中,可以使用标注和其它标号代替整型数。

预定义变量

表 A. 16 列出了可以在表达式中出现的特殊的变量。它们都是汇编器预定义的变量,不能被重复定义。

表 A. 16 预定义表达式

变 量	值
{ARCHITECTURE}	CPU 的 ARM 体系结构(“4T”为 ARMv4T)
{ARMASM_VERSION}	编译器版本号
{CONFIG} or {CODESIZE}	被编译指令的位数(ARM 状态 32 位,Thumb 状态 16 位)
{CPU}	被编译 CPU 的名字
{ENDIAN}	字节排列顺序(即大小端),big 或 little
{INTER}	如果 ARM/Thumb 互相作用开启,则为{TRUE}
{PC} (别名 .)	当前被编译指令的地址
{ROPI},{RWPI}	如果 read-only/read-write 位置独立,则为{TRUE}
{VAR} (别名 @)	MAP 计数器(见 MAP 保留字)

A. 4. 4 ARM 汇编保留字

下面是一个按照字母排列顺序给出的更普通 armasm 保留字列表。

ALIGN

ALIGN {<expression>{,<offset>}}

ARM 嵌入式系统开发

把下一条指令的地址对齐到 $q * \langle \text{expression} \rangle + \langle \text{offset} \rangle$ 处。因为对齐情况依赖于 ELF 段的开始情况, 所以它必须在适当的地方对齐(见 AREA 保留字)。 $\langle \text{expression} \rangle$ 必须是 2 的幂, 默认值是 4。如果没有指定 $\langle \text{offset} \rangle$, 则 $\langle \text{offset} \rangle$ 为 0。

AREA

AREA $\langle \text{section} \rangle \{, \langle \text{attr}_1 \rangle \} \{, \langle \text{attr}_2 \rangle \} \dots \{, \langle \text{attr}_k \rangle \}$

开启一个名为 $\langle \text{section} \rangle$ 的新的代码段或数据段。表 A.17 列出了可能的属性。

表 A.17 AREA 属性

属 性	含 义
ALIGN= $\langle \text{expression} \rangle$	ELF 段的对齐方式为 $2^{\langle \text{expression} \rangle}$ 字节对齐
ASSOC= $\langle \text{sectionname} \rangle$	如果这个段被连接, 则也要连接 $\langle \text{sectionname} \rangle$ 段
CODE	这个段定义为代码段而且是只读的
DATA	这个段定义为数据段而且是可读/写的
NOINIT	这个数据段不需要初始化
READONLY	这个段是只读的
READWRITE	这个段是可读/写的

ASSERT

ASSERT $\langle \text{logical-expression} \rangle$

汇编时断言(assert)。如果逻辑表达式为假, 则汇编以一个错误中止。

CN

$\langle \text{name} \rangle$ CN $\langle \text{numeric-expression} \rangle$

为一个协处理器的寄存器 $\langle \text{numeric-expression} \rangle$ 设置一个别名为 $\langle \text{name} \rangle$ 。

CODE16, CODE32

CODE16 指示汇编器把下面的指令汇编成 16 位 Thumb 指令。CODE32 指示汇编成 32 位的 ARM 指令(armasm 默认为 32 位)。

CP

$\langle \text{name} \rangle$ CP $\langle \text{numeric-expression} \rangle$

为一个协处理器 $\langle \text{numeric-expression} \rangle$ 设置一个别名为 $\langle \text{name} \rangle$ 。

DATA

<label> DATA

DATA 保留字指出标注 label 所指向的是数据而不是代码。在 Thumb 模式下,这可以防止连接器设置 label 的最低位。函数指针或者代码标注的位 0 对于 ARM 代码是 0,对于 Thumb 代码是 1(见 BX 指令)。

DCB,DCD{U},DCI,DCQ{U},DCW{U}

这些保留字按照表 A.18 分配一个或几个字节的存储器单元。每一个保留字后面都是一个以逗号隔开的初始化值的列表。如果指定了后缀 U,则汇编器不会做任何的对齐填充。

表 A.18 存储器初始化保留字

保留字	别名	数据大小/字节	初始化值
DCB	=	1	字节或字符串
DCW		2	16 位整型数(对齐到 2 字节)
DCD	&	4	32 位整型数(对齐到 4 字节)
DCQ		8	64 位整型数(对齐到 4 字节)
DCI		2 或 4	整型定义一个 ARM 或 Thumb 指令

例如:

```
hello      DCB      "hello".0
powers     DCD      1,2,4,8,10,0x20,0x40,0x80
           DCI      0xEA000000
```

ELSE (别名 |)

见 IF。

END

这个保留字必须出现在一个源文件的末尾。汇编器忽略在 END 保留字后面的内容。

ENDFUNC(别名 ENDP),ENDIF(别名)]

分别参见 FUNCTION 和 IF。

ENTRY

这个保留字为连接器指定了程序的入口点。这个入口点通常被包含在 ARM 的 C 库中。

EQU(别名 *)

```
<name> EQU    <numeric-expression>
```

这个保留字和 C 中的 #define 很类似。它定义一个标号 <name>, 值被表达式定义。这些值不能被重复定义。参看 A. 4.1 小节中关于可重复定义变量的使用。

EXPORT(别名 GLOBAL)

```
EXPORT    <symbol> {[WEAK]}
```

使用 EXPORT 声明一个全局标号, 可被其它目标文件和库文件连接引用。如果不使用这个命令, 则汇编标号只是这个目标文件的局部标号。可选项后缀 [WEAK] 暗示在使用这个标号之前, 要测试和解决其它实例引用这个标号。

EXTERN, IMPORT

```
EXTERN    <symbol> {[WEAK]}
```

```
IMPORT    <symbol> {[WEAK]}
```

2 个保留字都是声明一个在其它目标文件或库中定义的外部标号名。要使用该标号, 连接器会在连接时解释它。对于 IMPORT, 即使没有使用到该标号, 连接器也要解释它; 对于 EXTERN, 只有在使用到这个标号时, 连接器才会解释它。如果声明标号时带有可选项 [WEAK], 则即使连接器在连接处理时不能解释该标号也不会报错, 而是把该标号的值设为 0。

FIELD(别名 #)

参考 MAP。

FUNCTION(别名 PROC)和 ENDFUNC(别名 ENDP)

FUNCTION 和 ENDFUNC 保留字分别标记一个 ATPCS——compliant 函数的开始和结束。它们的主要用途是改进调试的范围, 在调试时可以将函数的调用进行回退, 即返回到函数调用前的状态。这也使性能分析器能够更加精确地分析出汇编函数的性能。必须在 FUNCTION 保留字前面给出 ATPCS (ARM Thumb Procedure Call Standard) 函数名。例如:

```
sub    FUNCTION
      SUB    r0, r0, r1
      MOV    pc, lr
      ENDFUNC
```

GBLA, GBLL, GBLS

分别用于全局的算术变量、逻辑变量、字符串变量的定义。参看 A. 4.1 小节。

GET

见 INCLUDE。

GLOBAL

见 EXPORT。

IF(别名[]),ELSE(别名[]),ENDIF(别名[])

这些保留字提供了汇编的条件语句。它们就像 C 语言中的 #if, #else, #endif。IF 保留字后面跟随着一个逻辑表达式。ELSE 保留字可以省略。例如：

```
IF ARCHITECTURE = "5TE"
    SMULBB    r0, r1, r1
ELSE
    MUL       r0, r1, r1
ENDIF
```

IMPORT

见 EXTERN。

INCBIN

```
INCBIN    <filename>
```

这个保留字在汇编程序中的当前位置原封不动地把二进制文件<filename>包含到当前文件中。例如, INCBIN table.dat。

INCLUDE(别名 GET)

```
INCLUDE    <filename>
```

使用这个保留字来包含其它的汇编文件。这和 C 中的 #include 命令类似。例如, INCLUDE header.h。

INFO(别名!)

```
INFO      <numeric_expression>, <string_expression>
```

如果<numeric_expression>非 0, 则汇编将会以一个<string_expression>的错误来中止; 否则汇编器将会以<string_expression>输出消息。

KEEP

```
KEEP      {<symbol>}
```

缺省情况下, 汇编器在目标文件中只包含全局标号(见 EXPORT), 而不包含局部标号。

使用 KEEP 来包含所有的局部标号或一个指定的局部标号。这对调试很有帮助。

LCLA, LCLL, LCLS

这些保留字分别声明一个宏的局部算术变量、局部逻辑变量和局部字符串变量。参看 A.4.1 小节。

LTORG

使用 LTOrg 插入一个文字池。汇编器使用文字池来存储出现在 LDR Rd, #<value> 指令中的常数。见 LDR 的第 19 条指令格式。通常汇编器会在程序的末尾自动插入一个文字池。但是如果一个程序太大,则 LDR 指令不能通过 pc 相对寻址(pc-relative addressing)访问该文字池,这时就需要在 LDR 指令附近人为地插入一个文字池。

MACRO, MEXIT, MEND

使用这些保留字来声明一个新的汇编宏或者伪指令。它的语法结构为:

```

MACRO
{ $ <arg_0> }    <macro_name>  { $ <arg_1> } {, $ <arg_2> } ... {, $ <arg_k> }
                  <macro_code>
MEND

```

宏参数被保存在哑变量(dummy variables) \$ <arg_i> 中。如果调用宏时没有提供某个参数,则该参数被设置成空串。MEXIT 保留字用于提前结束宏,这在 IF 语句后比较常见。例如,下面的宏定义了一个新的伪指令 SMUL,如果是 ARMv5TE 处理器,则进行一个 SMULBB 的运算;否则进行 MUL 运算。

```

MACRO
$ label          SMUL      $ a, $ b, $ c
                  IF {ARCHITECTURE} = "5TE"
$ label          SMULBB    $ a, $ b, $ c
                  MEXIT
                  ENDIF
$ label          MUL       $ a, $ b, $ c
MEND

```

MAP(别名 ^), FIELD(别名 #)

这些保留字定义一些类似于 C 语言中的结构体的对象。MAP 设置这个结构体的基地址或者偏移地址, FIELD 定义结构体的每个元素。语法结构如下:

```

MAP      <base> {, <base_register>}
name     FIELD    <field_size_in_bytes>

```

MAP 保留字设置专门的汇编器内置变量 {VAR} 为结构体的首地址。它的值为 $\langle \text{base} \rangle$ 或者寄存器相关的值 $\langle \text{base_register} \rangle + \langle \text{base} \rangle$ 。每一个 FIELD 保留字设置 $\langle \text{name} \rangle$ 的值为 VAR 的值, 然后按照指定的字节数增加 VAR 的值。对于寄存器相关的值, 表达式: INDEX; $\langle \text{name} \rangle$ 和; BASE; $\langle \text{name} \rangle$ 分别返回这个元素相对于基址寄存器的偏移量和基址寄存器的编号。

实际上使用 $\langle \text{base_register} \rangle$ 格式并不是很有用。可以使用普通格式, 然后在指令中清楚地指定基址寄存器。这样可以很方便地指向具有相同的数据类型、不同基址寄存器的结构体。下面的例子在堆栈上建立一个具有 2 个整型变量的结构体:

```

MAP      0                ;结构体元素偏移量为 0
count    FIELD 4          ;定义整型变量 count
type     FIELD 4          ;定义整型变量 type
size     FIELD 0          ;记录结构大小

SUB      sp, sp, #size    ;在堆栈里分配空间
MOV      r0, #0
STR      r0, [sp, #count] ;count 清 0
STR      r0, [sp, #type]  ;type 清 0

```

NOFP

这个保留字禁止在汇编文件中使用浮点运算。附录中没有涉及到浮点运算的指令和保留字。

OPT

这个保留字控制 armasm 的 -list 选项的格式。因为现在可以提供源代码级的调试, 所以很少使用它了。参看 armasm 文档。

PROC

见 FUNCTION。

RLIST, RN

```

<name>    RN      <numeric expression>
<name>    RLIST   <由 {} 括起来的 ARM 寄存器列表>

```

这 2 个保留字为一系列的 ARM 寄存器或一个 ARM 寄存器命名。例如, 下面的代码分别把 r0 命名为 arg, 把在 ATPCS 下受保护的寄存器命名为 saved。

```

arg        RN      0
saved      RLIST   {r4-r11}

```

ROUT

这个保留字定义一个新的局部标号使用范围。见 A.4.2 小节。

SETA, SETL, SETS

这些保留字分别设置算术变量、逻辑变量和字符串变量的值。见 A.4.1 小节。

SPACE(别名%)

```
{<label>} SPACE <numeric_expression>
```

这个保留字分配一段 <numeric_expression> 字节的内存单元。所有字节都初始化为 0。

WHILE, WEND

这 2 个保留字提供一段汇编时的循环。WHILE 后面跟着一个逻辑表达式。当这个表达式为真时,汇编器要重复汇编在 WHILE 和 WEND 段的代码。下面的例子演示了如何创建一个 1~65 536 的 2 的幂的数组。

```

        GBLA      count
count   SETA      1
        WHILE     count<= 65536
        DCD       count
count   SETA      2 * count
        WEND

```

608

A.5 GNU 汇编快速查询

本节概要地列出当以 GNU 汇编器、gas 作为 ARM 汇编器时,更有用的命令行和表达式。每一个汇编行都以如下的格式给出:

```
{<label>:} {<指令或保留字>} @ 注释
```

不像 ARM 汇编器需要对指令和保留字缩排。标号是通过后面有一个冒号而不是在一行的行首来识别的。下面的例子演示了一个简单的汇编文件,它定义了一个返回为 2 个输入变量的和的 add 函数:

```

.section      .text, "x"
.global      add

```

@ 给出外部连接的 add 标号

add;

ADD r0, r0, r1	@ 输入参数相加
MOV pc, lr	@ 从子函数返回

GNU 汇编保留字

下面是一个按照字母排列顺序给出的通用 gas 保留字列表。

.ascii “<string>”

把 string 当成数据插入汇编中,和 armasm 的 DCB 类似。

.asciz “<string>”

类似 .ascii,但在每个字符串后面跟一个零字节。

.balign <power_of_2> {, <fill_value> {, <max_padding>}}

对齐地址到<power_of_2>字节。汇编器通过添加<fill_value>字节或者默认值来对齐。如果需要填充的字节数大于<max_padding>,则对齐不会发生。

.byte <byte1> {, <byte2>}...

把一系列字节当成数据插入汇编,和 armasm 的 DCB 类似。

.code <number_of_bits>

按 bit 位设置指令的长度。16 位是 Thumb,32 位是 ARM。这和 armasm 的 CODE16 和 CODE32 类似。

.else

使用在 .if 和 .endif 之间。和 armasm 的 ELSE 类似。

.end

标记一个汇编文件的结束。这个通常被省略掉。

.endif

标志着条件汇编代码块的结束。见 .if, .ifdef, .ifndef。它和 armasm 的 ENDIF 类似。

.endm

结束一个宏定义。见 .macro。它和 armasm 的 MEND 类似。

.endr

结束一个循环。见 .rept 和 .irp。它和 armasm 的 WEND 类似。

.equ <symbol name>, <value>

设置一个标号(symbol)的值。它和 armasm 的 EQU 类似。

.err

以一个错误导致汇编的结束。

.exitm

从当前宏定义体中提前退出。见 macro。它和 armasm 的 MEXIT 类似。

.global <symbol>

给标号(symbol)一个外部连接。它和 armasm 的 EXPORT 类似。

.hword <short1>{, <short2>}...

把一系列的 16 位数当成数据插入汇编,和 armasm 的 DCW 类似。

.if <logical_expression>

定义一个条件代码块,以 .endif 结束。它和 armasm 的 IF 类似。

.ifdef <symbol>

如果<symbol>是定义了的,则包含(include)下面的一段代码块。这个代码块以 .endif 来结束。

.ifndef <symbol>

如果<symbol>是没有定义的,则包含(include)下面的一段代码块。这个代码块以 .endif 来结束。

.include "<filename>"

包含指定的源文件。它和 armasm 的 INCLUDE 或者和 C 的 #include 类似。

.irp <param> {, <val_1>} {, <val_2>}...

开启一个循环的代码块,块中每个 value 列表的 value 执行一次。块以一个 .endr 保留字来标记结束。在循环的代码块中,使用\<param>来代替 value 列表中的 value。

.macro <name> {<arg_1>} {, <arg_2>}...{, <arg_k>}

定义一个含有 k 个参数的名为<name>的宏。宏定义必须以 .endm 来标记结束。如果想提前跳出宏,则可使用 .exitm 保留字。这些保留字和 armasm 中的 MACRO, MEND 和 MEXIT 类似。必须在宏参数前加一个“\”。例如:

```
.macro    SHIFTLEFT  a, b
    .if    \b < 0
        MOV    \a, \a, ASR # -\b
```



```

        .exitm
    .endif
    MOV    \a, \a, LSL #\b
    .endm

```

.rept <number_of_times>

按照指定的次数重复执行一个代码块。这个块以 `.endr` 来标记结束。

<register_name> .req <register_name>

为一个寄存器取个名字。这和 `armasm` 的 `RN` 很类似,但是这里右边的寄存器不能只给出寄存器号,必须给出具体的寄存器,例如, `acc .req r0`。

.section <section_name> {, "<flags>"}

开始一个新的代码段或者数据段。通常,代码段称为 `.text`,一个经过初始化的数据段称为 `.data`,一个没有初始化的数据段称为 `.bss`。它们都有默认的标记(flag),连接器识别它们的默认名字。这个保留字和 `armasm` 的 `AREA` 类似。表 A.19 给出了 ELF 格式文件中 `<flag>` 字符串中可能出现的字符。

表 A.19 ELF 格式文件的 `.section` 标志

标 记	含 义
a	可分配段
w	可写段
x	可执行段

.set <variable_name>, <variable_value>

设置一个变量的值。它和 `armasm` 的 `SETA` 类似。

.space <number_of_byte> {, <fill_byte>}

生成给定数量的字节。如果指定了 `<fill_byte>`,则以指定的值填充每个字节;如果没有指定,则以 0 填充每个字节。它和 `armasm` 的 `SPACE` 类似。

.word <word1> {, <word2>}...

把一系列的 32 位字当成数据插入汇编,它和 `armasm` 的 `DCD` 类似。

附录 B

ARM 和 Thumb 指令编码

- ARM 指令集编码
- Thumb 指令集编码
- 程序状态寄存器

本附录给出 32 位 ARM 指令集和 16 位 Thumb 指令集的编码表,也对处理器状态寄存器 cpsr 和 spsr 的域进行了说明。

B.1 ARM 指令集编码

表 B.1 汇总了 ARMv6 体系结构 32 位 ARM 指令集的位编码。如果需要对 ARM 指令进行人工译码,则这张表是很有用的。为了有助于快速人工译码,已经对此表进行了一些扩充。对于表中没有列出的位映射,在 ARMv6 中是没有定义的,或者是不可预测的。

为了有效地使用表 B.1,可以采用以下译码步骤:

- 关注十六进制指令编码的第一个十六进制数字,即位 28~31。如果值为 0xF,则可直接跳到表 B.1 的结尾;否则这个十六进制数代表的是一个条件 cond。cond 的译码参考表 B.2。
- 整个表 B.1 的索引使用第 2 个十六进制数,即位 24~27(灰色部分)。
- 对于指令编码的位 4、位 7 或者位 23 用灰色标出,这些位也用作索引。
- 找到了表的正确入口后,查看 op 位。把 op 位置的二进制数字连接起来组成一个数字,以指示在表格左部由“|”分割的指令。例如,如果有 2 个 op 位,值是 1 和 0,二进制值 10 指出在表中的指令号为 2(第 3 条指令)。
- 指令操作数的命名与附录 A 中的描述相同。

表中使用以下缩写:

- 对于 STC 和 LDC 操作,如果有后缀 L,则 L 为 1。
- 如果 CPS 改变了处理器模式,则 M 为 1。处理器模式的定义在表 B.3。
- 在协处理器指令中,op1 和 op2 是操作码扩展域。
- post 说明是后变址寻址模式,例如[Rn],Rm 或者 [Rn],#immed。
- pre 说明是前变址寻址模式,例如[Rn,Rm] 或者 [Rn,#immed]。
- 如果寄存器 R_k 出现在寄存器列表中,则 register_list 是一个位域,且位 k 是置位的。
- rot 是字节循环,第 2 个操作数是 Rm ROR (8 * rot)。
- rotate 是位循环,第 2 个操作数是 #immed ROR (2 * rotate)。
- shift 和 sh 编码一个移位的类型和方向,参见表 B.4。
- U 用于选择寻址方式是向上(up)还是向下(down)。如果 U=1,则把偏移量加到基地址上,如[Rn],#4 或者[Rn,Rm];如果 U=0,那么就从基地址减去偏移量,如[Rn],#-4 或者[Rn],-Rm。
- unindexed 表示一个形式为 [Rn],{option} 的寻址模式。

表B.1 ARM指令译码表

指令类别(以op为索引)

AND EOR SUB RSB ADD ADC SBC RSC		cond	0	0	0	0	op		S	Rn	Rd	shift_size		shift	0	Rm	
AND EOR SUB RSB ADD ADC SBC RSC		cond	0	0	0	0	op		S	Rn	Rd	Rs		0 shift	1	Rm	
MUL		cond	0	0	0	0	0	0	S	Rd	0	0	0	0	1	Rm	
MLA		cond	0	0	0	0	0	0	1	S	Rd	Rn		Rs	1	Rm	
UMAAL		cond	0	0	0	0	0	1	0	0	RdHi	RdLo		Rs	1	Rm	
UMULL UMLAL SMULL SMLAL		cond	0	0	0	0	1	op		S	RdHi	RdLo		Rs	1	Rm	
STRH LDRH post		cond	0	0	0	0	U	0	0	op	Rn	Rd	0		0	Rm	
STRH LDRH post		cond	0	0	0	0	U	1	0	op	Rn	Rd	immed [7:4]		1	immed [3:0]	
LDRD STRD LDRSB LDRSH post		cond	0	0	0	0	U	0	0	op	Rn	Rd	0		0	Rm	
LDRD STRD LDRSB LDRSH post		cond	0	0	0	0	U	1	0	op	Rn	Rd	immed [7:4]		1	immed [3:0]	
MRS Rd, cpsr MRS Rd, spsr		cond	0	0	0	1	0	op	0	0	1	1	1	1	0	0	
MSR cpsr, Rm MSR spsr, Rm		cond	0	0	0	1	0	op	1	0	f	s	x	c	1	0	
BXJ		cond	0	0	0	1	0	0	1	0	1	1	1	1	0	0	
SMLAXy		cond	0	0	0	1	0	0	0	0	Rd	Rn		Rs	1	0	
SMLAWy		cond	0	0	0	1	0	0	1	0	Rd	Rn		Rs	1	0	
SMULWy		cond	0	0	0	1	0	0	1	0	Rd	0		0	0	0	
SMLALxy		cond	0	0	0	1	0	0	1	0	RdHi	RdLo		Rs	1	0	
SMULxy		cond	0	0	0	1	0	1	1	0	Rd	0		0	0	0	
TST TEQ CMP CMN		cond	0	0	0	1	0	op		1	Rn	0		0	0	0	
ORR BIC		cond	0	0	0	1	1	op	0	S	Rn	Rd	shift_size		shift	0	
MOV MVN		cond	0	0	0	1	1	op	1	S	0	0	0	0	0	0	
BX BLX		cond	0	0	0	1	0	0	1	0	1	1	1	1	0	0	
CLZ		cond	0	0	0	1	0	1	1	0	1	1	1	1	0	0	
QADD QSUB QDADD QDSUB		cond	0	0	0	1	0	op		0	Rn	Rd	0		0	0	
BKPT		1	1	1	0	0	0	0	1	0	immed[5:4]				0	1	1
TST TEQ CMP CMN		cond	0	0	0	1	0	op		1	Rn	0		0	0	0	
ORR BIC		cond	0	0	0	1	1	op	0	S	Rn	Rd	Rs		0 shift	1	
MOV MVN		cond	0	0	0	1	1	op	1	S	0	0	0	0	0	0	
SWP SWPB		cond	0	0	0	1	0	op	0	0	Rn	Rd	0		0	0	
STREX		cond	0	0	0	1	1	0	0	0	Rn	Rd	1		1	1	
LDREX		cond	0	0	0	1	1	0	0	1	Rn	Rd	1		1	1	

续表B.1

指令类别(以op为索引)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
cond					0	0	0	1	U	0	W	op		Rn		Rd		0	0	0	0	1	0	1								Rm	
cond					0	0	0	1	U	1	W	op		Rn		Rd																immed [3:0]	
cond					0	0	0	1	U	0	W	op		Rn		Rd		0	0	0	0	1	1	op	1							Rm	
cond					0	0	0	1	U	1	W	op		Rn		Rd																immed [3:0]	
cond					0	0	1	0	op		S			Rn		Rd																immed	
cond					0	0	1	1	0	op	1	0		f s x c	1	1	1	1														immed	
cond					0	0	1	1	0	op	1			Rn		0	0	0	0													immed	
cond					0	0	1	1	1	op	0	S		Rn		Rd																immed	
cond					0	0	1	1	1	op	1	S		0	0	0	0															immed	
cond					0	1	0	0	U	op	T	op		Rn		Rd																immed12	
cond					0	1	0	1	U	op	W	op		Rn		Rd																immed12	
cond					0	1	1	0	U	op	T	op		Rn		Rd																shift_size	shift_0
cond					0	1	1	0	0	op				Rn		Rd		1	1	1	1	0	0	1								Rm	
cond					0	1	1	0	0	op				Rn		Rd		1	1	1	1	0	0	1								Rm	
cond					0	1	1	0	0	op				Rn		Rd		1	1	1	1	0	1	1								Rm	
cond					0	1	1	0	0	op				Rn		Rd		1	1	1	1	0	1	1								Rm	
cond					0	1	1	0	0	op				Rn		Rd		1	1	1	1	1	1	1								Rm	
cond					0	1	1	0	1	0	0	0		Rn		Rd																shift_size	op_0
cond					0	1	1	0	1	op	1			immed5		Rd																shift_size	sh_0
cond					0	1	1	0	1	op	1	0		immed4		Rd		1	1	1	1	0	0	1								Rm	
cond					0	1	1	0	1	0	0	0		Rn		Rd		1	1	1	1	0	1	1								Rm	
cond					0	1	1	0	1	op	1	1		1	1	1	1															op_0	1
cond					0	1	1	0	1	op	0	0		Rn!=1111		Rd		rot	0	0	0	1	1	1								Rm	
cond					0	1	1	0	1	op	0	0		1	1	1	1															Rm	
cond					0	1	1	0	1	op	1	0		Rn!=1111		Rd		rot	0	0	0	1	1	1								Rm	
cond					0	1	1	0	1	op	1	0		1	1	1	1															Rm	
cond					0	1	1	0	1	op	1	1		Rn!=1111		Rd		rot	0	0	0	1	1	1								Rm	
cond					0	1	1	0	1	op	1	1		1	1	1	1															Rm	
cond					0	1	1	1	U	op	W	op		Rn		Rd																shift_size	shift_0
cond					0	1	1	1	0	0	0	0		Rd		Rn!=1111																Rs	0 op X
cond					0	1	1	1	0	0	0	0		Rd		1	1	1	1													Rs	0 op X
cond					0	1	1	1	0	1	0	0		RdHi		RdLo																Rs	0 op X
cond					0	1	1	1	0	1	0	0		RdLo																		Rs	0 op X

STRH|LDRH pre

STRH|LDRH pre

LDRD|STRD|LDRSB|LDRSH pre

LDRD|STRD|LDRSB|LDRSH pre

AND|EOR|SUB|IRSB|

ADD|ADC|SV|RSC

MSR qsr, #imm|MSR spsr, #imm

TST|TEQ|CMP|CMN

ORR|BIC

MOV|MVN

STR|LDR|STRB|LDRB post

STR|LDR|STRB|LDRB pre

STR|LDR|STRB|LDRB post

{|S|Q|S|H|} |U|U|Q|U|H|ADD|16

{|S|Q|S|H|} |U|U|Q|U|H|ADDSUBX

{|S|Q|S|H|} |U|U|Q|U|H|SUBADDX

{|S|Q|S|H|} |U|U|Q|U|H|SUB16

{|S|Q|S|H|} |U|U|Q|U|H|ADD8

{|S|Q|S|H|} |U|U|Q|U|H|SUB8

PKBT|PKHTB

{|S|U|S|AT

{|S|U|S|AT|16

SEL

REV |REV16| |REVSH

{|S|U|X|TAB|16

{|S|U|X|TAB

{|S|U|X|TAB|16

{|S|U|X|TB

{|S|U|X|TAH

{|S|U|X|TH

STR|LDR|STRB|LDRB pre

SMLAD|SMLSD

SMUAD|SMUSD

SMLALD|SMLSLD

STRH|DRH pre

STRH|DRH pre

LDRD|STRD|LDRSB|DRSH pre

LDRD|STRD|LDRSB|DRSH pre

AND|EOR|SUB|RSB|

ADD|ADC|SBV|RSC

MSR cpsr, #imm|MSR spsr, #imm

TST|TEQ|CMP|CMN

ORR|BIC

MOV|MVN

STR|DR|STRB|DRB post

STR|DR|STRB|DRB pre

STR|DR|STRB|DRB post

{ISIQ|SH|IU|QU|UH|ADD|16

{ISIQ|SH|IU|QU|UH|ADDSUBX

{ISIQ|SH|IU|QU|UH|SUBADDX

{ISIQ|SH|IU|QU|UH|SUB16

{ISIQ|SH|IU|QU|UH|ADD8

{ISIQ|SH|IU|QU|UH|SUB8

PKBT|PKHTB

{SIU|SAT

{SIU|SAT16

SEL

REV|REV16|REVSH

{SIU|XTAB16

{SIU|XTAB

{SIU|XTB16

{SIU|XTB

{SIU|XTAH

{SIU|XTH

STR|DR|STRB|DRB pre

SMLAD|SMLSD

SMUAD|SMUSD

SMLALD|SMLSLD

续表B.1

指令类别(以op为索引)

SMMLA ||| SMMMLS
 SMMUL
 USADA8
 USAD8
 Undefined and expected to stay so
 STMDA|LDMDA|STMA|LDMA
 STMDB|LDMDB|STMB|LDMIB
 B to instruction_address+8*4*offset
 BL to instruction_address+8*4*offset
 MRR|MRRC
 STC{L}|LDC{L} unindexed
 STC{L}|LDC{L} post
 STC{L}|LDC{L} pre
 CDP
 MCR|MRC
 SWI
 CPS|CPSIE|CPSID
 SETEND LE|SETEND BE
 PLD pre
 PLD pre
 RFEDA|RFEDA|RFEDB|RFEDB
 SRSDA|SRSDA|SRSDB|SRSDB
 BLX instruction+8*4*offset+2*a
 MRR2|MRRC2
 STC2{L}|LDC2{L} unindexed
 STC2{L}|LDC2{L} post
 STC2{L}|LDC2{L} pre
 CDP2
 MCR2|MRC2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond
0	1	1	1	1	0	1	0	1	0	1	0	1	Rd	Rn!=1111	Rs	op	R	1	Rm												
0	1	1	1	1	0	1	0	1	0	1	0	1	Rd	1	1	1	1	Rs	0	0	R	1	Rm								
0	1	1	1	1	1	0	0	0	0	0	0	0	Rd	Rn!=1111	Rs	0	0	0	1	Rm											
0	1	1	1	1	1	0	0	0	0	0	0	0	Rd	1	1	1	1	Rs	0	0	0	1	Rm								
0	1	1	1	1	1	1	1	1	1	1	1	1		x																	
cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond	cond
1	0	0	0	0	op	^	W	op					Rn					register list													
1	0	0	1	op	^	W	op						Rn					register list													
1	0	1	0											signed 24-bit branch offset																	
1	0	1	1											signed 24-bit branch offset																	
1	1	0	0	0	0	1	0	op					Rn	Rd	copro	op1		Cm													
1	1	0	0	1	L	0	op						Rn	Cd	copro	option															
1	1	0	0	U	L	1	op						Rn	Cd	copro	immed8															
1	1	0	0	U	L	W	op						Rn	Cd	copro	immed8															
1	1	1	0					op1					Cn	Cd	copro	op2	0	Cm													
1	1	1	0					op1	op				Cn	Rd	copro	op2	1	Cm													
1	1	1	1											immed24																	
1	1	1	1	0	0	0	1	0	0	0	0	op	M	0	0	0	0	0	0	a	i	f	0	mode							
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	op	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	0	1	U	1	0	1		Rn	1	1	1	1	immed12													
1	1	1	1	0	1	1	1	U	1	0	1		Rn	1	1	1	1	shift size	shift	0	Rm										
1	1	1	1	1	0	0	op	op	0	W	1		Rn	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	op	op	1	W	0	1	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	1	a							signed 24-bit branch offset																	
1	1	1	1	1	1	0	0	0	1	0	op		Rn	Rd	copro	op1		Cm													
1	1	1	1	1	1	0	0	1	L	0	op		Rn	Cd	copro	option															
1	1	1	1	1	1	0	0	U	L	1	op		Rn	Cd	copro	immed8															
1	1	1	1	1	1	0	1	U	L	W	op		Rn	Cd	copro	immed8															
1	1	1	1	1	1	1	1		op1				Cn	Cd	copro	op2	0	Cm													
1	1	1	1	1	1	1	1	0	op1	op			Cn	Cd	copro	op2	1	Cm													

表 B.2 cond 译码表

二进制	十六进制	cond	二进制	十六进制	cond
0000	0	EQ	1000	8	HI
0001	1	NE	1001	9	LS
0010	2	CS/HS	1010	A	GE
0011	3	CC/LO	1011	B	LT
0100	4	MI	1100	C	GT
0101	5	PL	1101	D	LE
0110	6	VS	1110	E	{AL}
0111	7	VC			

- 如果指令出现后缀 R(round,舍入),则 R 为 1。
- 如果 Load-store 指令中出现后缀 T,则 T 为 1。
- 如果“!”(writeback,回写)在指令助记符中出现,则 W 为 1。
- 如果指令出现后缀 X(exchange,交换),则 X 为 1。
- 如果出现后缀 B,则 x 和 y 是 0;如果出现后缀 T,则 x 和 y 为 1。
- 如果 LDM 或 STM 指令有后缀“ \cdot ”,则“ \cdot ”为 1。

表 B.3 mode 译码表

二进制	十六进制	mode
10000	0x10	用户模式(_usr)
10001	0x11	快速中断模式(_fiq)
10010	0x12	中断模式(_irq)
10011	0x13	监控模式(_svc)
10111	0x17	中止模式(_abt)
11011	0x1B	未定义模式(_und)
11111	0x1F	系统模式

表 B.4 shift, shift_size 和 Rs 译码表

shift	shift_size	Rs	移位操作
00	0~ 31	N/A	LSL # shift_size
00	N/A	Rs	LSL Rs
01	0	N/A	LSR # 32
01	0~ 31	N/A	LSL # shift_size
01	N/A	Rs	LSL Rs
10	0	N/A	ASR # 32
10	1~ 31	N/A	ASR # shift_size
10	N/A	Rs	ASR Rs
11	0	N/A	RRX
11	1~ 31	N/A	ROR # shift_size
11	N/A	Rs	ROR Rs
N/A	0~ 31	N/A	值 shift 是隐式的:对于 PKHBT 是 00; 对于 PKHTB 是 10; 对于 SAT 是 2 * sh

B.2 Thumb 指令集编码

表 B.5 汇总了 16 位 Thumb 指令集的位编码。如果需要对 Thumb 指令进行人工译码,则这张表是很有用的。为了有助于快速人工译码,已经对此表进行了一些扩充。此表包括了到体系结构 THUMBv3 的指令定义。表中没有列出的位映射,在 THUMBv3 中是没有定义的,或是不可预测的。

为了有效地使用表 B.5,可以使用以下的译码流程:

- 使用指令编码的第一个十六进制数(位 12~15,灰色的)来索引表格。
- 使用灰色标出的位 0~11 进行索引。
- 找到了表的正确入口后,查看 op 位。把 op 位置的二进制数字连接起来组成一个数字,以指示在表格左部由“|”分割的指令。例如,如果有 2 个 op 位,值是 1 和 0,则二进制值 10 指出在表中的指令号为 2(第 3 条指令)。
- 指令操作数的命名与附录 A 中的描述相同。

表中使用以下缩写:

- 如果寄存器 Rk 出现在寄存器列表中, 则 register_list 是一个位域, 且位 k 是置位的。
- 如果 lr 出现在 PUSH 操作的寄存器列表中, 或者 pc 出现在 POP 操作的寄存器列表中, 则 R=1。

表B.5 Thumb指令译码表

指令类别(以op为索引)

LSL|LSR

ASR

ADD|SUB

ADD|SUB

MOV|CMP

ADD|SUB

AND|EOR|LSL|LSR

ASR|ADC|SBC|ROR

TST|NEG|CMP|CMN

ORR|MUL|BIC|MVN

CPY Ld, Lm

ADD|MOV Ld, Hm

ADD|MOV Hd, Lm

ADD|MOV Hd, Hm

CMP

CMP

CMP

BX|BLX

LDR LD, [pc, #immed*4]

STR|STRH|STRB|LDRSB pre

LDR|LDRH|LDRB|LDRSH pre

STR|LDR LD, [Ln, #immed*4]

STRB|LDRB LD, [Ln, #immed]

STRH|LDRH Ld, [Ln, #immed*2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	op	immed5					Lm			Ld		
0	0	0	1	0	immed5					Lm			Ld		
0	0	0	1	1	0	op	Lm			Ln			Ld		
0	0	0	1	1	1	op	immed3			Ln			Ld		
0	0	1	0	op	Ld/Ln			immed8							
0	0	1	1	op	Ld			immed8							
0	1	0	0	0	0	0	0	op	Lm/Ls			Ld			
0	1	0	0	0	0	0	1	op	Lm/Ls			Ld			
0	1	0	0	0	0	1	0	op	Lm			Ld/Ln			
0	1	0	0	0	0	1	1	op	Lm			Ld			
0	1	0	0	0	1	1	0	0	0	Lm			Ld		
0	1	0	0	0	1	op	0	0	1	Hm & 7			Ld		
0	1	0	0	0	1	op	0	1	0	Lm			Hd & 7		
0	1	0	0	0	1	op	0	1	1	Hm & 7			Hd & 7		
0	1	0	0	0	1	0	1	0	1	Hm & 7			Ln		
0	1	0	0	0	1	0	1	1	0	Lm			Hd & 7		
0	1	0	0	0	1	0	1	1	1	Hm & 7			Hd & 7		
0	1	0	0	0	1	1	1	op	Rm				0 0 0		
0	1	0	0	1	Ld			immed8							
0	1	0	0	0	op	Lm			Ln			Ld			
0	1	0	1	1	op	Lm			Ln			Ld			
0	1	1	0	op	immed5					Ln			Ld		
0	1	1	1	op	immed5					Ln			Ld		
1	0	0	0	op	immed5					Ln			Ld		

续表B.5

指令类别(以op为索引)

STR|LDR LD, [sp, #immed*4]

ADD Ld, pc, #immed*4|
ADD LD, sp, #immed*4ADD sp, #immed*4|SUB sp,
#immed*4

SXTB|SXTB|UXTH|UXTB

REV|REV16|REVSH

PUSH|POP

SETEND LE|SETEND BE

CPSIE|CPSID

BKPT immed8

STMIA|LDmia Ln!, {register-list}

B<cond> instruction address+
4+offset*2

未定义, 保留。

SWI immed8

B instruction address+4+offset*2

BLX ((instruction+4+
(poff<12)+offset*4) & ~3
前面必须有一条分支前缀指令。分支前缀指令, 后面必须跟
BL或BLX指令。BL instruction+4+(poff<12)+offset*2
前面必须有一条分支前缀指令。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	op	Ld		immed8								
1	0	1	0	op	Ld		immed8								
1	0	1	1	0	0	0	0	op	immed7						
1	0	1	1	0	0	1	0	op	Lm		Ld				
1	0	1	1	1	0	1	0	op	Lm		Ld				
1	0	1	1	op	1	0	R	register_list							
1	0	1	1	0	1	1	0	0	1	0	1	op	0	0	0
1	0	1	1	0	1	1	0	0	1	1	op	0	a	i	f
1	0	1	1	1	1	1	0	immed8							
1	1	0	0	op	Ln		register_list								
1	1	0	1	cond<1110			signed 8-bit offset								
1	1	0	1	1	1	1	0	x							
1	1	0	1	1	1	1	1	immed8							
1	1	1	0	0	signed 11-bit offset										
1	1	1	0	1	unsigned 10-bit offset										0
1	1	1	1	0	signed 11-bit prefix offset poff										
1	1	1	1	1	unsigned 11-bit offset										

B.3 程序状态寄存器

表 B.6 列出了 ARMv6 的 32 位程序状态寄存器的位域定义。

621

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N				Z	C	V	Q	Res		J	Res			GE[3:0]				Res				E		A	I		F	T	mode		

域	用途
N	负标志,位 31,记录标志设置操作的结果
Z	零标志,如果标志设置操作的结果是 0,则置位
C	进位标志,记录无符号加法溢出、减法无借位、循环移位。参见表 A.3
V	溢出标志,记录标志设置操作的有符号溢出
Q	饱和标志,一些指令在饱和时设置该位。参见附录 A 中 QADD 指令(ARMv5E 和以后的体系结构)
J	J=1 说明 Java 执行(必须 T=0),使用 BXJ 指令改变该位(ARMv5J 及以后的体系结构)
Res	这些位是保留的,用于以后的扩展。软件不应操作这些位
GE[3:0]	SIMD 大于或等于标志,参看附录 A 中的 SADD(ARMv6)
E	控制数据大小端。参看附录 A 中的 SETEDN(ARMv6)
A	A=1 禁止不明确的数据中止(ARMv6)
I	I=1 禁止 IRQ 中断
F	F=1 禁止 FIQ 中断
T	T=1 表示 Thumb 状态;T=0 表示 ARM 状态。使用 BX 或者 BLX 指令改变该位(ARMv4T 和以后的架构)
mode	当前处理器模式,参看表 B.4

附录 C

处理器与体系结构

- ARM 命名规则
- 内核与体系结构

C 处理器与体系结构

本附录中列出 ARM 处理器的命名、相应的 ARM 内核名称以及指令集架构(ISA)。ARM7TDMI 以前的处理器没有列出。

例如,表 C.3 显示了 ARM966E-S 处理器是 ARM9E 核,实现的 ARM 体系结构版本是 5TE。所以任何 ARMv5TE 的二进制代码都可以在 ARM966E-S 上执行。

C.1 ARM 命名规则

所有的 ARM 处理器都使用一个共同的命名规则。当然,这个规则随着时间的推移也在不断地发展。*ARM 内核名称*的形式为:ARM{x}{label}。这里 *x* 是 ARM 核的编号;label 是一些字母组成的,用于表示其它的特征,在表 C.1 中有描述。*ARM 处理器名称*的形式为:ARM{x}{y}{z}{labels},这里 *y* 和 *z* 用于定义处理器 cache 大小和内存管理模式。表 C.2 列出了 ARM 处理器编号的规则。

表 C.1 Label 属性

属 性	说 明
D	ARM 核支持通过 JTAG 接口调试(debug)。ARMv5 及以后的 ARM 核都自动包含 D
E	ARM 核在 ARMv5 上增加 DSP 指令扩展。ARMv5 及以后的内核都自动包含 E
F	ARM 核通过向量浮点(VFP)结构支持硬件浮点
I	ARM 核通过嵌入式 ICE 单元(EmbeddedICE cell)支持硬件断点和观察点。ARMv5 及其后的 ARM 核都自动包含 I
J	ARM 核支持 Jazelle Java 加速体系结构
M	ARM 核支持 ARMv3 的长乘法指令。ARMv4 及以后的内核都自动包含 M
-S	ARM 核使用可综合的硬件设计
T	ARM 核对 ARMv4 及以后的体系结构支持 Thumb 指令集。ARMv6 及以后的内核都自动包含 T

表 C.2 ARM 处理器编号:ARM{x}{y}{z}.

<i>x</i>	<i>y</i>	<i>z</i>	说 明	示 例
7	*	*	ARM7 处理器核	ARM7TDMI
9	*	*	ARM9 处理器核	ARM926EJ-S
10	*	*	ARM10 处理器核	ARM1026EJ-S
11	*	*	ARM11 处理器核	ARM1136J-S

续表 C.2

x	y	z	说 明	示 例
*	2	*	cache 和 MMU	ARM920T
*	3	*	物理地址标记的 cache 和 MMU	ARM1136J-S
*	4	*	cache 和 MPU	ARM946E-S
*	6	*	写缓冲,但无 cache	ARM966E-S
*	*	0	标准 cache 大小	ARM920T
*	*	2	缩小的 cache	ARM922T
*	*	6	包括紧密耦合 SRAM(TCM)	ARM946E-S

标号(labels),或者属性,通常包含在随时间变化的不同体系结构版本中。例如,在 ARMv4 体系结构的处理器中,标号 T 表示包含了 Thumb。但是由于在 ARMv5 及以后的处理器中都包含 Thumb,所以在此后的版本中,就没有必要再标出 T 了。

C.2 内核与体系结构

表 C.3 列出了每种 ARM 处理器对应的 ARM 核以及处理器使用的体系结构版本。

表 C.3 处理器、内核和体系结构版本

处理器产品	处理器核	ARM ISA	Thumb ISA	VFP ISA
ARM7TDMI	ARM7TDMI	v4T	v1	
ARM7TDMI-S	ARM7TDMI-S	v4T	v1	
ARM7EJ-S	ARM7EJ	v5TEJ	v2	
ARM740T	ARM7TDMI	v4T	v1	
ARM720T	ARM7TDMI	v4T	v1	
ARM920T	ARM9TDMI	v4T	v1	
ARM922T	ARM9TDMI	v4T	v1	
ARM940T	ARM9TDMI	v4T	v1	
Intel SA-110	StrongARM1	v4		
ARM926EJ-S	ARM9EJ	v5TEJ	v2	
ARM946E-S	ARM9E	v5TE	v2	
ARM966E-S	ARM9E	v5TE	v2	
ARM1020E	ARM10E	v5TE	v2	

续表 C.3

处理器产品	处理器核	ARM ISA	Thumb ISA	VFP ISA
ARM1022E	ARM10E	v5TE	v2	
ARM1026EJ - S	ARM10EJ	v5TEJ	v2	
Intel XScale™	XScale	v5TE	v2	
ARM1136J - S	ARM11	v6J	v3	
ARM1136JF - S	ARM11	v6J	v3	v2

附录 D

指令周期定时

- 指令周期定时表的使用
- ARM7TDMI 指令周期定时
- ARM9TDMI 指令周期定时
- StrongARM1 指令周期定时
- ARM9E 指令周期定时
- ARM10E 指令周期定时
- Intel XScale 指令周期定时
- ARM11 指令周期定时

本附录列出了一些通用 ARM 实现上的指令周期定时数。在不同的实现版本上,指令周期定时数可能会有所不同,定时数也会受到一些外部事件的影响,如中断、存储器速度和 cache 失效等。本附录给出的数据仅供参考,性能的验证还应在实际的硬件平台上进行。最新的指令周期定时信息请参考制造商的数据手册。

ARM 核使用流水线技术实现,所以一条指令执行实际占用的周期数,可能与其前/后执行的指令都有关。在优化代码时,要明确这些指令之间的相互影响,请注意在定时表中标注有“说明(Notes)”的那些指令。

D.1 指令周期定时表的使用

可以使用以下步骤来计算指令执行的周期数。

- ① 使用附录 C 中的表 C.3 找到所使用的 ARM 核。例如,ARM7xx 通常是 ARM7TDMI 核;ARM9xx 使用 ARM9TDMI 核;ARMxxE 使用 ARM9E 核。
- ② 在本附录中找到所使用的 ARM 核对应的指令周期定时表。
- ③ 在表的左边找到相关的指令类别。ALU 类指令是算术指令和逻辑指令的简称,包括:ADD,ADC,SUB,RSB,SBC,RSC,AND,ORR,BIC,EOR,CMP,CMN,TEQ,MOV,MVN,CLZ 等。
- ④ 在“周期(cycles)”列中得到相应的数值。这个数值是指令通常需要占用的周期数,这里假定条件码匹配并且不考虑与其它指令的相关。周期计数可能与表 D.1 中的某个缩写有关。
- ⑤ 如果“说明”列中包含形式为“+k if condition”的注释,则须在原来的指令周期计数上,加上这些额外的周期。
- ⑥ 检查能够导致处理器暂停的互锁条件。经常会出现这样的情况:当前指令要使用前面指令的结果,而此时结果还没有准备好。除非有特殊说明,在指令执行的第一个周期,输入寄存器就需要被使用,而输出结果要到指令执行的最后一个周期结束后才有效。但是,具有多个执行阶段的流水线实现,就会较早地需要输入操作数,而结果可能要在几个周期后才输出。表 D.2 对描述这些内容的“说明”部分所使用的语句进行了解释。
- ⑦ 如果指令的条件码不匹配,则其不会被执行,通常这也要占用一个周期。但是对于有些体系结构的实现,即使指令没有执行,也可能要占用多个周期。可以参考形式为“[k cycles if not execute]”的说明。

表 D.1 标准的周期缩写

缩 写	含 义
B	协处理器发出的忙-等待周期数,依赖于协处理器的设计
M	乘法器迭代周期数,依赖于寄存器 R_s 中的值。每一个具体的实现都包含一张表,以说明如何根据 R_s 计算 M
N	在多寄存器 load-store 指令中要传输的字的数目。如果 pc 出现在寄存器列表中,则 N 也包含 pc。 N 至少是 1

表 D.2 流水线操作说明

说明语句	含 义
Rd is not available for k cycles	指令的结果寄存器 Rd 要在指令结束 k 个周期后才可以作为其它指令的输入。如果试图过早地使用 Rd,则内核会暂停,直到 k 个周期结束之后才有效
Rn is required k cycles early	在指令开始执行之前,指令的输入寄存器 Rn 就必须已经有效 k 个周期;否则,内核将会等到条件满足才开始执行
Rn is not required until the kth cycle	输入寄存器 Rn 不是在指令执行的第一个周期就被读取 cycle,而是在第 k 个周期读取。因此如果 Rn 在那个时刻有效,则 ARM 内核就不会暂停
You cannot start a type X	指令使用的资源也被一个 X 类型的指令使用,而且当前指令将会在执行结束后继续使用资源 k 个周期
Instruction for k cycle	如果试图在 k 个周期结束前执行一个 X 类型的指令,则内核将暂停直至 k 个周期结束

D.2 ARM7TDMI 指令周期定时

ARM7TDMI 内核是 3 级流水线结构,其中包含一个执行阶段。一条指令占用的周期数通常不依赖于前/后继指令。乘法电路使用一个可提前终止的 32×8 位的乘法器阵列。乘法迭代的周期数 M 依赖于寄存器 R_s 的值,参见表 D.3。表 D.4 给出了 ARM7TDMI 指令周期定时数。

表 D.3 ARM7TDMI 的提前终止乘法器*

M	Rs 范围(使用第一个适用的范围)	Rs 位格式	s=符号位	x=通配符位	
1	$-2^8 \leq x < 2^8$	88888888	88888888	88888888	XXXXXXXX
2	$2^{16} \leq x < 2^{16}$	88888888	88888888	XXXXXXXX	XXXXXXXX
3	$2^{24} \leq x < 2^{24}$	88888888	XXXXXXXX	XXXXXXXX	XXXXXXXX
4	其余的 x	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

表 D.4 ARM7TDMI(ARMv4T)指令周期定时数

指令类别	周期数	说 明
ALU	1	如果使用寄存器指定的移位量来对 Rs 移位,则周期数加 1 如果 Rd 是 pc,则周期数加 2
B,BL,BX	3	
CDP	1+B	
LDC	1+B+N	
LDR/B/H/SB/SH	3	如果 Rd 是 pc,则指令周期数加 2
LDM	2+N	如果 pc 出现在寄存器列表中,则周期数加 2
MCR	2+B	
MLA	2+M	
xMLAL	3+M	
MRC	3+B	
MRS,MSR	1	
MUL	1+M	
XMULL	2+M	
STC	1+B+N	
STR/B/H	2	
STM	1+N	
SWI	3	
SWP/B	4	

* 提前终止乘法器按照表 D.3 中 1,2,3,4 的顺序来判断输入值 x 的范围,以最短的时间完成相应数据范围的乘法操作。——译者注

D.3 ARM9TDMI 指令周期定时

ARM9TDMI 内核基于一条具有 1 个执行阶段、2 个存储器操作阶段的 5 级流水线。在 load 指令执行后,通常会有 1~2 个周期的延迟,才可以使用装载的结果。在 load 指令后立即使用装载的数据,会增加互锁周期。乘法器电路使用可以提前终止的 32×8 位乘法器阵列。乘法迭代周期 M 依赖于寄存器 R_s 的值,参见表 D.5。表 D.6 给出 ARM9TDMI 的指令周期定时数。

表 D.5 ARM9TDMI 的提前终止乘法器

M	R_s 范围(使用第一个适用的范围)	R_s 位格式	s =符号位	x =通配符位	
1	$-2^8 \leq x < 2^8$	ssssssss	ssssssss	ssssssss	xxxxxxxx
2	$-2^{16} \leq x < 2^{16}$	ssssssss	ssssssss	xxxxxxxx	xxxxxxxx
3	$-2^{24} \leq x < 2^{24}$	ssssssss	xxxxxxxx	xxxxxxxx	xxxxxxxx
4	其余的 x	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

表 D.6 ARM9TDMI(ARMv4T)指令周期定时数

指令类别	周 期	说 明
ALU	1	如果使用寄存器指定的移位量来对 R_s 移位,则周期数加 1 如果 R_d 是 pc,则周期数加 2。
B,BL,BX	3	
CDP	$1+B$	
LDC	$B+N$	
LDRB/H/SB/SH	1	R_d 在 2 个周期内不能使用
LDR R_d not pc	1	R_d 在 1 个周期内不能使用
LDR R_d is pc	5	
LDM not loading pc	N	如果 $N=1$ 或者最后一个装载的寄存器在下一个周期中被使用,则周期数加 1
LDM loading pc	$N+4$	
MCR	$1+B$	
MRC R_d not pc	$1+B$	R_d 在 1 个周期内不能使用
MRC R_d is pc	$3+B$	

续表 D.6

指令类别	周 期	说 明
MRS	1	
MSR	1	如果 csx 域的任何位被更新,则指令周期数加 2
MUL, MLA	$2+M$	
xMULL, xMLAL	$3+M$	
STC	$B+N$	
STR/B/H	1	
STM	N	如果 $N=1$,则指令周期数加 1
SWI	3	
SWP/B	2	Rd 在 1 个周期内不能使用

D.4 StrongARM1 指令周期定时

StrongARM1 核基于一条 5 级流水线。在使用 load 或者乘法指令后,通常会有 1 个周期的延迟,结果才可以被使用。另外,如果在前一条乘法指令执行后立即启动另外一条乘法指令,则也会有 1 个周期的延迟。乘法器电路使用可提前终止的 32×12 位乘法器阵列。乘法迭代周期 M 依赖于寄存器 Rs 的值,参见表 D.7。表 D.8 给出 StrongARM1 的指令周期定时数。

表 D.7 StrongARM1 的提前终止乘法器

M	Rs 范围(使用第一个适用的范围)	Rs 位格式	s =符号位	x =通配符位	
1	$-2^{11} \leq x < 2^{11}$	SSSSSSSS	SSSSSSSS	SSSSSXXX	XXXXXXXX
2	$-2^{23} \leq x < 2^{23}$	SSSSSSSS	SSSSSSSS	SSSSSSSXX	XXXXXXXX
3	其余的 x	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

表 D.8 StrongARM1(ARMv4)指令周期定时数

指令类别	周 期	说 明
ALU	1	如果使用寄存器指定的移位,则即使指令没有执行,周期数也要加 1 如果 Rd 是 pc,并且指令执行,则周期数加 2
B, BL	2	
LDR/B/H Rd not pc	1	Rd 在 1 个周期内不能使用
LDRSB/SH Rd not pc	2	Rd 在 1 个周期内不能使用
LDR Rd is pc	4	
LDM N=1, not pc	2	[如果没有执行,则 2 个周期]
LDM N>1, not pc	N	最后一个装载的值在 1 个周期内不可使用 [如果没有执行,则 N 个周期]
LDM loading pc	N+3	[如果没有执行,则 max(N,2)个周期]
MRS	1	Rd 在 1 个周期内不能使用
MSR to cpar	3	如果 csx 域的任何位被更新,则指令周期数加 1
MSR to spsr	1	
MUL, MLA	M	Rd 在 1 个周期内不能使用,并且在下一个周期不能启动其它的乘法
MULS, MLAS	4	
xMULL, xMLAL	1+M	Rd 在 1 个周期内不能使用,并且在下一个周期不能启动其它乘法 [如果指令没有执行,则指令周期数为 2]
xMULLS, xMLALS	5	[如果指令没有执行,则指令周期数为 2]
STR/B/H	1	
STM	N	如果 N=1,则指令周期数加 1 [如果指令没有执行,则使用的指令周期数相同]
SWP/B	2	[如果指令没有执行,则使用周期数为 2]

D.5 ARM9E 指令周期定时

ARM9E 内核基于一条 5 级流水线。在使用 load 或者乘法指令后,通常会有 1~2 个周期的延迟,结果才可以被使用。乘法器电路使用 32×16 位的乘法器阵列,乘法器不会提前

终止。表 D.9 给出了 ARM9E 的指令周期定时数。

表 D.9 ARM9Erev2(ARMv5TE) 指令周期定时数

指令类别	周 期	说 明
ALU Rd not pc	1	如果使用寄存器指定的移位,则周期数加 1
ALU Rd is pc	3	如果指令是逻辑操作或者使用任何移位操作,则周期数加 1
B,BL,BX,BLX	3	
CDP	$1+B$	
LDC	$B+N$	
LDRB/H/SH/SH	1	Rd 在 2 个周期内不能使用 如果装载偏移使用移位,则周期数加 1
LDR Rd not pc	1	Rd 在 1 个周期内不能使用 如果装载偏移使用移位,则周期数加 1
LDR Rd is pc	5	如果装载偏移使用移位,则周期数加 1
LDRD	2	R(d+1)在 1 个周期内不可使用
LDM ,not loading pc	N	如果 $N=1$ 或者最后一个装载寄存器在下一个周期中被使用,则指令周期数加 1
LDM loading pc	$N+4$	
MCR	$1+B$	
MCRR	$2+B$	
MRC Rd is not pc	$1+B$	Rd 在 1 个周期内不能使用
MRC Rd is pc	$4+B$	
MRRC	$2+B$	Rd 在 1 个周期内不能使用
MRS	2	
MSR	1	如果 csx 域的任何值被更新,则指令周期数加 2
MUL,MIA	2	Rd 在 1 个周期内不能使用,除非是作为乘累加的累加器输入
MULS,MIAS	4	
xMULL,xMLAL	3	RdHi 在 1 个周期内不能使用,除非是作为乘累加的累加器输入
xMULLS,xMLALS	5	
PLD	1	
QxADD,QxSUB	1	Rd 在 1 个周期内不能使用

续表 D.9

指令类别	周 期	说 明
SMULxy, SMLAxy, SMLWx, SMLAWx	1	Rd 在 1 个周期内不能使用, 除非是作为乘累加的累加器输入
SMLALxy	2	RdHi 在 1 个周期内不能使用, 除非是作为乘累加的累加器输入
STC	$B+N$	
STR/B/H	1	如果使用移位的偏移量, 则指令周期数加 1
STRD	2	
STM	N	如果 $N=1$, 则指令周期数加 1
SWI	3	
SWP/B	2	Rd 在 1 个周期内不能使用

D.6 ARM10E 指令周期定时

ARM10E 内核基于一条带分支预测的 5 级流水线。在使用 load 或者乘法指令后, 通常会有 1 个周期的延迟, 结果才可以被使用。ARM10E 使用 64 位的数据总线, 所以 load-store 指令每周期可以传输 64 位。乘法器不会提前终止。表 D.10 给出了 ARM10E 的指令周期定时数。

表 D.10 ARM10E(ARMv5TE)指令周期定时数

指令类别	周 期	说 明
ALU	1	如果使用寄存器指定的移位或者使用 RRX, 则周期数加 1; 如果 Rd 是 pc, 则指令周期数加 4 一个例外是 MOV pc, Rn, 占用 4 个周期
B, BX	0~2	如果分支预测错误(mispredicted), 则指令周期数加 4
BL, BLX	1~2	如果分支预测错误(mispredicted), 则指令周期数加 4
CDP	1	
LDC	1	结果的产生依赖于协处理器
LDR/B/H/SB/SH	1	Rd 在 1 个周期内不能使用
Rd not pc		如果寻址方式是带有(常量)移位选项的寄存器前变址寻址, 则指令周期数加 1

续表 D.10

指令类别	周 期	说 明
LDR Rd is pc	6	如果偏移量(前变址或后变址)是一个移位的寄存器,则周期数加 1 [如果没有执行,则周期数为 2]
LDRD	1	Rd 和 R(d+1)在 1 个周期内不可使用
LDM ,not loading pc	1	第一个数据项在 1 个周期不可使用。一旦地址是 8 字节对齐的,数据就可以被成对装载——每周期 2 个数据。所以,第 k 个数据项应该在 $(k+a+1)/2$ 个周期后才有效,这里 a 是基地址的位 2。在这个指令完成前,不能启动另外的 load 或 store 指令
LDM loading pc	$L+6$	$L=(N+a)/2$, a 是基地址的位 2
MCR, MCRR	1	
MCR{R}C Rd not pc	1	Rd 在 1 个周期内不能使用
MRC Rd is pc	2	
MRS	1	
MSR to cpsr	1	如果 csx 域的任何值被更新,则指令周期数加 3
MSR to spsr	3	如果指令没有执行,则指令周期数为 2
MUL, MLA	2	Rd 在 1 个周期内不能使用
MULS, MLAS	4	
xMULL, xMLAL	3	RdHi 在 1 个周期内不能使用
xMULLS, xMLALS	5	
PLD	1	如果使用移位的寄存器作偏移量,则指令周期数加 1
QxADD, QxSUB	1	Rd 在 1 个周期内不能使用
SMUL _{xy} , SMULW _x	1	Rd 在 1 个周期内不能使用
SMLA _{xy} , SMLAW _x	2	RdHi 在 1 个周期内不能使用
SMLAL _{xy}	2	
STC	1	
STR/B/H	1	如果使用前变址寻址的移位寄存器作偏移量,则指令周期数目加 1
STRD	1	
STM	1	一旦地址是 8 字节对齐的,1 个周期就可以存储 2 个寄存器。寄存器列表中的寄存器在被存储之前,不可对其改写。在本指令执行完之前,不能启动另外的 load 或 store 指令
SWP/B	2	

D.7 Intel XScale 指令周期定时

Intel XScale 基于一条 7 级流水线。在执行 load 指令后,通常会有 2 个周期的延迟,结果才可以被使用。乘法指令的发射通常有固定的周期数,但是结果需要经过若干周期后才能被使用,具体的周期数依赖于寄存器 Rs 的值。表 D.11 显示了乘法迭代周期数 M 和 Rs 值的关系。表 D.12 列出了 Intel XScale 的指令周期定时数。

表 D.11 Intel XScale 的提前终止乘法器

M	Rs 范围(使用第一个适用的范围)	Rs 位格式	s= 符号位	x= 通配符位	
1	$-2^{15} \leq r < 2^{15}$	SSSSSSSS	SSSSSSSS	XXXXXXXX	XXXXXXXX
2	$-2^{27} \leq r < 2^{27}$	SSSSSXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
3	其余的 r	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

表 D.12 Intel XScale(ARMv5TE) 指令周期定时数

指令类别	周 期	说 明
ALU	1	如果使用寄存器指定的移位或者使用 RRX,则指令周期数加 1;如果 Rd 是 pc,则指令周期数加 4
B,BL	1	如果分支预测错误(mispredicted),则指令周期数加 4
BX,BLX	5	[如果没有执行,则需要 1 个周期]
LDR/B/H/SB/SH	1	Rd 在 2 个周期内不能使用
Rd not pc		
LDR Rd is pc	8	[如果没有执行,则需要 2 个周期]
LDRD	1	Rd 在 2 个周期内不可使用,R(d+1)在 3 个周期内不可使用
LDM ,not loading pc	2+N	最后一个装载的值在 2 个周期内不可使用,倒数第 2 个装载的值在 1 个周期内不可使用
LDM loading pc	7+N	如果 $N < 3$,则指令使用 10 个周期 [如果没有执行指令,则需要 3+N 个周期]
MCR to copro 15	2	
MRC from copro 15	4	
MRS	1	Rd 在 1 个周期内不能使用
MSR	2	如果 csx 域的任何值被更新,则指令周期数加 4

续表 D.12

指令类别	周 期	说 明
MUL, MLA	1	Rd 在 M 个周期内不能使用, 在 $M-1$ 个周期内不能启动另一条乘法指令
MULS, MLAS	$1+M$	
xMULL	1	RdHi 在 $M+1$ 个周期内不能使用, RdLo 在 M 个周期内不可使用, 在 M 个周期内不能启动另一条乘法指令
xMLAL	2	RdHi 在 M 个周期内不能使用, RdLo 在 $M-1$ 个周期内不可使用, 在 $M-1$ 个周期内不能启动另一条乘法指令
xMULLS, xMLALS	$2+M$	
PLD	1	
QxADD, QxSUB	1	Rd 在 1 个周期内不能使用
SMULxy, SMLAxy	1	Rd 在 1 个周期内不能使用
SMULWx, SMLAWx,	1	Rd 在 2 个周期内不能使用, 在 1 个周期内不能启动另一条乘法指令
SMLALxy	2	RdHi 在 1 个周期内不能使用
STR/B/H	1	
STRD	2	
STM	$2+N$	
SWI	6	
SWP/B	5	

D.8 ARM11 指令周期定时

ARM11 内核基于一条具有 3 个执行阶段的 8 级流水线。在 load 指令执行后, 通常有 2 个周期的延迟, 装载的数据才可以被使用。一些操作, 比如移位、乘法和地址计算等, 要求其输入寄存器提前 1 个周期有效。

例如, 下面的代码将会使内核中止 3 个周期, 因为 load 指令的结果在 2 个周期内不可使用, 而移位指令要求输入寄存器提前 1 个周期有效。

```
LDR    r0, [r1]           ; r0 在 2 个周期内不可使用
MOV    r2, r0, ASR#3      ; 要求 r0 提前 1 个周期有效
```

ARM11 有一个单独的地址发生器, 可以在 1 个周期内计算出简单的地址。更复杂的

地址计算需要 2 个周期。表 D. 13 定义了在不同寻址方式下,地址计算所需要的周期数⁴。

表 D. 13 ARM11 地址计算周期

A	寻址模式	
1	[Rn, # <signed-offset>]{!}	
	[Rn], # <signed-offset>	
	[Rn, Rm{, LSL # 2}]{!}	
	[Rn], Rm{, LSL # 2}	
2	[Rn, - Rm]{!}	
	[Rn], - Rm	
	[Rn, {-} <shifted_Rm>]{!}	移位操作不是 LSL # 0 或 LSL # 2
	[Rn, {-} <shift_Rm>	移位操作不是 LSL # 0 或 LSL # 2

ARM11 内核使用分支预测技术,因此能把由于程序流程改变而产生的延时周期最小化。置位 CP15 寄存器 c1 的位 11,可以使能分支预测。有 3 种分支预测器:

- **静态预测器** 预测没有记录在分支预测 cache 中的相对跳转。这种情况是处理器第一次发现一个给定的分支指令。静态预测器向前按条件满足、跳转发生方向预测,向后按条件不满足、跳转不发生方向预测。
- **动态预测器** 预测已经记录在分支预测 cache 中的相对跳转。分支预测 cache 有 128 个基于分支指令跳转地址的项。每个 cache 项预测分支目标和跳转是否发生。一个 cache 项有 4 个状态:几乎不发生(strongly not taken)、基本不发生(weakly not taken)、很少发生(weakly taken)、经常发生(strongly taken)。每当跳转发生时,状态右移一个(如果可以的话);每当跳转没有发生时,状态左移一个(如果可以)。
- **返回堆栈** 预测无条件的子程序返回指令。堆栈有 3 个项保存着从 3 个最深层次的 BL 和 BLX 子程序调用的返回地址。

表 D. 14 列出了 ARM11(ARMv6)指令周期定时数。

表 D. 14 ARM11(ARMv6)指令周期定时数

指令类别	周 期	说 明
ALU Operations except a MOV to pc (for MOV to pc, see BX)	1	如果 Rm 执行一个常量移位,则应提前 1 个周期有效 如果使用寄存器指定移位量,则指令周期数加 1 这种情况下,Rs 要求提前 1 个周期,Rn 在第 2 个周期才会使用 如果 Rd 是 pc,则指令周期数加 6

续表 D.14

指令类别	周 期	说 明
B <immed> BL <immed> BLX <immed>	1	假设动态预测成功。一些动态预测的分支跳转可能被重叠(在流水线执行中),从而使用周期数为 0 如果成功地静态预测,则指令周期数加 3 如果动态预测或者静态预测没有成功,则指令周期数加 4,在这种情况下,标志需要提前 2 个周期
BX lr MOV pc,lr	4	如果无条件并且返回堆栈为空,则指令周期数加 1 如果无条件并且返回堆栈没有预测,则指令周期数加 3 如果条件执行指令周期加 1,这时标志要求提前 2 个周期
BX Rm (not lr) BLX Rm MOV pc, Rm (not lr)	5	如果 MOV 指令没有用到移位,并且是条件执行的,那么标志要提前 2 个周期 如果 MOV 使用常量移位,则指令周期数加 1,并且 Rm 要提前 1 个周期,Rn 在第 2 个周期才会使用。如果是条件执行,则标志需要提前 1 个周期 如果 MOV 使用寄存器移位,则 Rs 需要提前 1 个周期,Rn 在第 2 个周期才会使用
CPS	1	如果模式切换,则指令周期数加 1
LDR/B/H/SB/SH/D Rd not pc	A	Rd 在 2 个周期内不能使用。对于 LDRD 和 R(d+1),2 个周期不可使用 如果装载是潜在的、没有对齐的(基址或者偏移没有对齐),则在下一个指令周期不能启动另外的存储访问指令 如果装载没有对齐,则对于 LDR/H/SH,Rd 在 3 个周期内不能使用;对于 LDRD,Rd 在 2 个周期内不能使用,R(d+1)在 3 个周期内不能使用
LDR pc,[sp,#off](!) LDR pc,[sp],#off LDR pc not using a Constant stack offset	4 A+7	如果无条件并且返回堆栈为空,则指令周期数加 4 如果无条件并且返回堆栈没有预测,则指令周期数加 5 如果条件执行,则周期数加 2
LDM ,not loading pc	1	在接下来的 $(N+a-1)/2$ 个周期内不能进行存储访问,这里 a 是地址的第 2 位,第 k 个寄存器在 $(k+a+3)/2$ 个周期内不能访问

指令类别	周 期	说 明
LDM sp({}) loading pc	4	如果条件执行或者返回堆栈为空或者返回堆栈没有预测,则指令周期数加 5。在 $(N+a)$ 个周期内不能进行其它的存储访问,第 k 个寄存器在 $(k+a+5)$ 个周期内不可使用
LDM loading pc not from the stack	8	在 $(N+a)/2$ 个周期内不能进行其它的存储访问,第 k 个寄存器列表中的寄存器在 $(k+a+5)/2$ 个周期内不能使用
MCR/MCRR	1	和存储器访问的情形相同
MRC/MRRC	1	和存储器访问的情形相同,结果寄存器在 2 个周期内不可使用
MRS	1	
MSR to cpsr	1	如果任何 csx 域更新,则指令周期数加 3
MSR to spsr	5	
MUL, MLA	2	Rd 在 2 个周期内不能使用。如果作为另外一个乘法指令的累加器输入,则此时它在 1 个周期内不可使用 Rm 和 Rs 要提前 1 个周期,对于 MLA 指令, Rn 在第 2 个周期才会使用
MULS, MLAS	5	Rm 和 Rs 要提前 1 个周期,对于 MLAS 指令, Rn 在第 2 个周期才会使用
xMULL, xMLAL	3	RdLo 在 1 个周期内不可使用。RdHi 在 2 个周期内不可使用。如果这些寄存器作为另外一个乘法器的累加器,则延迟周期减少 1 个。Rm 和 Rn 要提前 1 个周期可以使用 RdLo 对于 MLAL 指令,在第 2 个周期才会使用
xMULLS, xMLALS	6	Rm 和 Rs 要提前 1 个周期可以使用, RdLo 对于 MLAL 指令,在第 2 个周期才会使用
PKHBT, PKHTB	1	Rm 要提前 1 个周期可用
PLD	A	
QxADD, QxSUB	1	Rd 在 1 个周期内不能使用,对于 QDADD 和 QDSUB 指令, Rn 要提前 1 个周期可用
REV, REV16, REVSH	1	Rm 要提前 1 个周期可用
{S, SH, Q, U, UH, UQ}		对于饱和或二分操作(带 SH, Q, UH, UQ 前缀的)
ADD16, ADDSUBX, SUBADDX, SUB16, ADD8, SUB8		Rd 在 1 个周期内不可使用,对于 ADDSUBX 和 SUBADDX 指令, Rm 要提前 1 个周期可用

续表 D.14

指令类别	周 期	说 明
SEL	1	
SETEND	1	
SMULxy, SMLAxy, SMULWy, SMLAWy, SMUAD, SMLAD, SMUSD, SMLSD	1	Rd 在 2 个周期内不能使用, 除非作为其它乘累加的累加器输入, 这时 Rd 在 1 个周期内不可使用 Rm 和 Rs 要提前 1 个周期可以使用
SMLALxy, SMLALD{X} SMLSLD{X}	2	RdLo 在 1 个周期内不能使用, RdHi 在 2 个周期内不能使用。作为其它乘法操作的累加器输入, 会减少 1 个周期的延迟。Rm 和 Rs 要提前 1 个周期可用, RdHi 在第 2 个周期才使用
SMMUL{R}, SMMLA{R}, SMMLS{R}	2	Rd 在 2 个周期内不能使用, 除非作为其它乘累加的累加器输入, 这时 Rd 在 1 个周期内不可使用 Rm 和 Rs 要提前 1 个周期可以使用, Rn 在第二个周期才使用
SSAT, USAT, SSAT16, USAT16	1	Rd 在 1 个周期内不可使用, 对于 SSAT 和 USAT 指令, Rm 要提前 1 个周期可以使用
STR/B/H/D	A	如果装载是潜在的、没有对齐的(基址或者偏移没有对齐), 则在下一个指令周期不能启动另外的存储访问指令 对于 STRD, 在 1 个周期内不能写 R(d+1)
STM	1	在随后的 $(N+a-1)/2$ 个周期内, 不能启动存储访问指令, a 是地址的第 2 位。在 $k/2$ 个周期内不能写寄存器列表中的第 k 个寄存器
SWI	8	
SWP/B	2	Rd 在 1 个周期内不能使用
SXT, UXT	1	Rm 要提前 1 个周期可以使用
UMAAL	3	RdLo 在 1 个周期内不能使用, RdHi 在 2 个周期内不能使用, 作为其它乘法操作的累加器输入, 会减少 1 个周期的延迟。Rm 和 Rs 要提前 1 个周期可用, RdLo 在第 2 个周期才使用
USAD8, USADA8	1	Rd 在 2 个周期内不能使用, 除非 USAD8 的结果作为 USADA8 的累加器, 这样只有 1 个周期的延迟 Rm 和 Rs 要提前 1 个周期可用

附录 E

建议的参考读物

- ARM 参考
- 算法参考
- 存储器管理与 cache 体系结构(硬件描述与参考)
- 操作系统参考

E.1 ARM 参考

- ARM Architecture Reference Manual, Second Edition, Published 2001, edited by David Seal. Addison-Wesley. The definitive reference for the ARM architecture definition.
- ARM System-on-Chip Architecture, Second Edition, Published 2000, by Steve Furber. Addison-Wesley. Covers the hardware aspects of ARM processors and SOC design.

E.2 算法参考

- Digital Signal Processing: Principles, Algorithms, and Applications, by John G. Proakis and Dimitris G. Manolakis. Published 1996. PrenticeHall. This is a solid book on DSP algorithms.
- The Art of Computer Programming: Seminumerical Algorithms, by Donald E. Knuth. Third Edition, Published 1998. Addison-Wesley. A highly respected work covering random number generation, algorithms used for extended-precision arithmetic, as well as many other fundamental algorithms.

643

E.3 存储器管理与 cache 体系结构(硬件综述与参考)

- The Cache Memory Book, by Jim Handy. Second Edition (1998). Academic Press. Provides a detailed discussion of cache design.
- Computer Architecture: A Quantitative Approach, by John L. Hennessy et al. Morgan Kaufmann. 2nd edition (1996). A classic text on computer hardware design.
- Computer Organization and Design: The Hardware/software Interface, by David A. Patterson et al. 1997. Morgan Kaufmann. A solid textbook showing the relationship between hardware and software in modern computer systems.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernel.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.

E.4 操作系统参考

- Design of the UNIX Operating System, by Maurice J. Bach(1986). Prentice-Hall. Describes the internal algorithms and structures of the UNIX System V kernal.
- Operating Systems, 2nd edition(1990) by Harvey M. Deitel. Addison-Wesley. A very good introductory text on operating systems.
- Modern Operating Systems, 2nd edition(2001) by Andrew Tanenbaum. Prentice-Hall. A thorough overview of operating system design.